

PEARSON



· 新 · 锐 · 编 · 程 · 语 · 言 · 集 · 萃 ·

Programming in Go

Creating Applications for the 21st Century

Go语言程序设计

【英】Mark Summerfield 著

许式伟 吕桂华 徐立 何李石 译



人民邮电出版社
POSTS & TELECOM PRESS

目 录

[封面](#)

[扉页](#)

[版权](#)

[版权声明](#)

[译者序](#)

[致谢](#)

[引言](#)

[第1章 5个例子](#)

[1.1 开始](#)

[1.2 编辑、编译和运行](#)

[1.3 Hello Who?](#)

[1.4 大数字——二维切片](#)

[1.5 栈——自定义类型及其方法](#)

[1.6 americanise示例——文件、映射和闭包](#)

[1.7 从极坐标到笛卡儿坐标——并发](#)

[1.8 练习](#)

[第2章 布尔与数值类型](#)

[2.1 基础](#)

[2.2 布尔值和布尔表达式](#)

[2.3 数值类型](#)

[2.3.1 整型](#)

[2.3.2 浮点类型](#)

[2.4 例子: statistics](#)

[2.4.1 实现一个简单的统计函数](#)

[2.4.2 实现一个基本的HTTP服务器](#)

[2.5 练习](#)

第3章 字符串

3.1 字面量、操作符和转义

3.2 比较字符串

3.3 字符和字符串

3.4 字符串索引与切片

3.5 使用fmt包来格式化字符串

3.5.1 格式化布尔值

3.5.2 格式化整数

3.5.3 格式化字符

3.5.4 格式化浮点数

3.5.5 格式化字符串和切片

3.5.6 为调试格式化

3.6 其他字符处理相关的包

3.6.1 strings包

3.6.2 strconv包

3.6.3 utf8包

3.6.4 unicode包

3.6.5 regexp包

3.7 例子: m3u2pls

3.8 练习

第4章 集合类型

4.1 值、指针和引用类型

4.2 数组和切片

4.2.1 索引与分割切片

4.2.2 遍历切片

4.2.3 修改切片

4.2.4 排序和搜索切片

4.3 映射

[4.3.1 创建和填充映射](#)

[4.3.2 映射查询](#)

[4.3.3 修改映射](#)

[4.3.4 键序遍历映射](#)

[4.3.5 映射反转](#)

[4.4 例子](#)

[4.4.1 猜测分隔符](#)

[4.4.2 词频统计](#)

[4.5 练习](#)

[第5章 过程式编程](#)

[5.1 语句基础](#)

[5.1.1 类型转换](#)

[5.1.2 类型断言](#)

[5.2 分支](#)

[5.2.1 if语句](#)

[5.2.2 switch语句](#)

[5.3 for循环语句](#)

[5.4 通信和并发语句](#)

[5.5 defer、panic和recover](#)

[5.6 自定义函数](#)

[5.6.1 函数参数](#)

[5.6.2 init\(\)函数和main\(\)函数](#)

[5.6.3 闭包](#)

[5.6.4 递归函数](#)

[5.6.5 运行时选择函数](#)

[5.6.6 泛型函数](#)

[5.6.7 高阶函数](#)

[5.7 例子：缩进排序](#)

[5.8 练习](#)

[第6章 面向对象编程](#)

[6.1 几个关键概念](#)

[6.2 自定义类型](#)

[6.2.1 添加方法](#)

[6.2.2 验证类型](#)

[6.3 接口](#)

[6.4 结构体](#)

[6.5 例子](#)

[6.5.1 FuzzyBool——一个单值自定义类型](#)

[6.5.2 Shapes——一系列自定义类型](#)

[6.5.3 有序映射——一个通用的集合类型](#)

[6.6 练习](#)

[第7章 并发编程](#)

[7.1 关键概念](#)

[7.2 例子](#)

[7.2.1 过滤器](#)

[7.2.2 并发的Grep](#)

[7.2.3 线程安全的映射](#)

[7.2.4 Apache报告](#)

[7.2.5 查找副本](#)

[7.3 练习](#)

[第8章 文件处理](#)

[8.1 自定义数据文件](#)

[8.1.1 处理JSON文件](#)

[8.1.2 处理XML文件](#)

[8.1.3 处理纯文本文件](#)

[8.1.4 处理Go语言二进制文件](#)

[8.1.5 处理自定义的二进制文件](#)

[8.2 归档文件](#)

[8.2.1 创建zip归档文件](#)

[8.2.2 创建可压缩的tar包](#)

[8.2.3 解开zip归档文件](#)

[8.2.4 解开tar归档文件](#)

[8.3 练习](#)

[第9章 包](#)

[9.1 自定义包](#)

[9.1.1 创建自定义的包](#)

[9.1.2 导入包](#)

[9.2 第三方包](#)

[9.3 Go命令行工具简介](#)

[9.4 Go标准库简介](#)

[9.4.1 归档和压缩包](#)

[9.4.2 字节流和字符串相关的包](#)

[9.4.3 容器包](#)

[9.4.4 文件和操作系统相关的包](#)

[9.4.5 图像处理相关的包](#)

[9.4.6 数学处理包](#)

[9.4.7 其他一些包](#)

[9.4.8 网络包](#)

[9.4.9 反射包](#)

[9.5 练习](#)

[附录A 后记](#)

[附录B 软件专利的危害](#)

[附录C 精选书目](#)

Go语言程序设计
Programming in Go
Creating Applications for the 21st Century
【英】Mark Summerfield 著
许式伟 吕桂华 徐立 何李石 译

人民邮电出版社
北京

图书在版编目 (CIP) 数据

GO语言程序设计/ (英) 萨默菲尔德 (Summerfield,M.) 著; 许式伟等译.--北京: 人民邮电出版社, 2013.8

(新锐编程语言集萃)

书名原文: Programming in Go

ISBN 978-7-115-31790-2

I.①G... II.①萨...②许... III.①程序语言—程序设计 IV.①TP312

中国版本图书馆CIP数据核字 (2013) 第099941号

内容提要

本书既是一本实用的Go语言教程, 又是一本权威的Go语言参考手册。书中从如何获取和安装Go语言环境, 以及如何建立和运行Go程序开始, 逐步介绍了Go语言的语法、特性以及一些标准库, 内置数据类型、语句和控制结构, 然后讲解了如何在Go语言中进行面向对象编程, Go语言的并发特性, 如何导入和使用标准库包、自定义包及第三方软件包, 提供了评价Go语言、以Go语言思考以及用Go语言编写高性能软件所需的所有知识。

本书的目的是通过使用语言本身提供的所有特性以及Go语言标准库中一些最常用的包, 向读者介绍如何进行地道的Go语言编程。本书自始至终完全从实践的角度出发, 每一章提供多个生动的代码示例和专门设计的动手实验, 帮助读者快速掌握开发技能。本书适合对Go语言感兴趣的各个层次的Go语言程序员阅读和参考。

◆著 [英] Mark Summerfield

译 许式伟 吕桂华 徐立 何李石

责任编辑 杨海玲

责任印制 程彦红 杨林杰

◆人民邮电出版社出版发行 北京市崇文区夕照寺街14号

邮编 100061 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

北京艺辉印刷有限公司印刷

◆开本: 800×1000 1/16

印张: 23.25

字数: 523千字 2013年8月第1版

印数: 1-3500册 2013年8月北京第1次印刷

著作权合同登记号 图字: 01-2012-6496号

定价: 69.00元

读者服务热线: (010)67132692 印装质量热线: (010)67129223

反盗版热线: (010)67171154

广告经营许可证: 京崇工商广字第0021号

版权声明

Authorized translation from the English language edition, entitled: Programming in Go, 978-0-321-77463-7 by Mark Summerfield, published by Pearson Education, Inc., publishing as Addison-Wesley Professional, Copyright © 2012 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. and POSTS & TELECOM PRESS
Copyright © 2013.

本书中文简体字版由Pearson Education Asia Ltd. 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

译者序

关注过我的人可能都知道，我在新浪微博、《Go语言编程》一书中都非常高调地下了一个论断：Go语言将超过C、Java，成为未来十年最流行的语言。

为什么我可以如此坚定地相信，选择Go语言不会有错，并且相信Go语言会成为未来10年最流行的语言？除了Go语言的并发编程模型深得我心外，Go语言的各种语法特性显得那么深思熟虑、卓绝不凡，其对软件系统架构的领悟，让我深觉无法望其项背，处处带给我惊喜。

Go语言给我的第一个惊喜是大道至简的设计哲学。

Go语言是非常简约的语言。简约的意思是少而精。少就是指数级的多。Go语言极力追求语言特性的最小化，如果某个语法特性只是少写几行代码，但对解决实际问题的难度不会产生本质的影响，那么这样的语法特性就不会被加入。Go语言更关心的是如何解决程序员开发上的心智负担。如何减少代码出错的机会，如何更容易写出高品质的代码，是Go设计时极度关心的问题。

Go语言追求显式表达。任何封装都是有漏洞的，最佳的表达方式就是用最直白的表达方式，所以也有人称Go语言为“所写即所得”的语言。

Go语言也是非常追求自然（nature）的语言。Go不只是提供极少的语言特性，并极力追求语言特性最自然的表达，也就是这些语法特性被设计成恰如多少人期望的那样，尽量避免惊异。事实上，Go语言

的语法特性上的争议是非常少的。这些也让Go语言的入门门槛变得非常低。

Go语言给我的第二个惊喜是最对胃口的并行支持。

我对服务端开发的探索，始于Erlang语言，并且认为Erlang风格并发模型的精髓是轻量级进程模型。然而，Erlang除了语言本身不容易被程序员接受外，其基于进程邮箱做消息传递的并发编程模型也小有瑕疵。我曾经在C++中实现了一个名为CERL的网络库，刚开始在C++中完全模仿Erlang风格的并发编程手法，然而在我拿CERL库做云存储服务的实践中，发现了该编程模型的问题所在并做了相应的调整，这就是后来的CERL 2.0版本。有意思的是，CERL 2.0与Go语言的并行编程思路不谋而合。某种程度上来说，这种默契也是我创办七牛时，Go语言甚至语法特性都还没有完全稳定，我们技术选型就坚决地采纳了Go语言的重要原因。

Go语言给我的第三个惊喜是接口。

Go语言的接口，并非是你看到在Java和C#中的接口，尽管看起来有点像。Go语言的接口是非侵入式的接口，具体表现在实现一个接口不需要显式地进行声明。不过，让我意外的不是Go的非侵入式接口。非侵入式接口只是我接受Go语言的基础。在接口（或契约）的表达上，我一直认为Java和C#这些主流的静态类型语言都走错了方向。C++的模板尽管机制复杂，但是走在了正确的方向上。C++0x（后来的C++11）呼声很高的concept提案被否，着实让不少人伤了心。但Go语言的接口远不是非侵入式接口那么简单，它是Go语言类型系统的纲，这表现在以下几个方面。

（1）只要某个类型实现了接口要的方法，那么我们就说该类型实现了此接口。该类型的对象可赋值给该接口。

（2）作为1的推论，任何类型（包括基础类型如bool、int、string等）的对象都可以赋值给空接口interface{}。

(3) 支持接口查询。如果你曾经是Windows程序员，你会发现COM思想在Go语言中通过接口优雅呈现。并且Go语言吸收了其中最精华的部分，而COM中对象生命周期管理的负担，却因为Go语言基于gc方式的内存管理而不复存在。

Go语言给我的第四个意外惊喜是极度简化但完备的面向对象编程(OOP)方法。

Go语言废弃大量的OOP特性，如继承、构造/析构函数、虚函数、函数重载、默认参数等；简化的符号访问权限控制，将隐藏的this指针改为显式定义的receiver对象。Go语言让我看到了OOP编程核心价值原来如此简单——只是多数人都无法看透。

Go语言带给我的第五个惊喜是它的错误处理规范。

Go语言引入了内置的错误(error)类型以及defer关键字来编写异常安全代码，让人拍案叫绝。下面这个例子，我在多个场合都提过：

```
f, err := os.Open(file)
if err != nil {
    ...// 错误处理
    return
}
defer f.Close()
...// 处理文件数据
```

Go语言带给我的第六个惊喜是它功能的内聚。

一个最典型的案例是Go语言的组合功能。对于多数语言来说，组合只是形成复合类型的基本手段，这一点只要想想C语言的struct就清楚了。但Go语言引入了匿名组合的概念，它让其他语言原本需要引入继承这样的新概念来完成事情，统一到了组合这样的一个基础上。

在C++中，你需要这样定义一个派生类：

```
class Foo : public Base {
```

```
...  
};
```

在Go语言中你只要：

```
type Foo struct {  
    Base  
    ...  
}
```

更有甚者，Go语言的匿名组合允许组合一个指针：

```
type Foo struct {  
    *Base  
    ...  
}
```

这个功能可以实现C++中一个无比晦涩难懂的特性，叫“虚拟继承”。但同样的问题换成从组合角度来表达，直达问题的本质，清晰易懂。

Go语言带给我的第七个惊喜是消除了堆与栈的边界。

在Go语言之前，程序员清楚地知道哪些变量在栈上，哪些变量在堆上。堆与栈是基于现代计算机系统的基础工作模型上形成的概念，Go语言屏蔽了变量定义在堆上还是栈上这样的物理结构，相当于封装了一个新的计算机工作模型。这一点看似与Go语言显式表达的设计哲学不太一致，但我个人认为这是一项了不起的工作，而且与Go语言的显式表达并不矛盾。Go语言强调的是对开发者的程序逻辑（语义）的显式表达，而非对计算机硬件结构的显示表达。对计算机硬件结构的高度抽象，将更有助于Go语言适应未来计算机硬件发展的变化。

Go语言带给我的第八个惊喜是Go语言对C语言的支持。

可以这么说，Go语言是除了Objective-C、C++这两门以兼容C为基础目标的语言外的所有语言中，对C语言支持最友善的一个。什么

语言可以直接嵌入C代码？只有Go。什么语言可以无缝调用C函数？只有Go。对C语言的完美支持，是Go快速崛起的关键支撑。还有比C语言更让人觊觎的社区财富吗？那是一个取之不尽的金矿。

总而言之，Go语言是一门非常具有变革性的语言。尽管40年（从1970年C语言诞生开始算起）来出现的语言非常之多，各有各的特色，让人眼花缭乱。但是我个人固执地认为，谈得上突破了C语言思想，将编程理念提高到一个新高度的，仅有Go语言而已。

Go语言很简单，但是具备极强的表现力。从目前的状态来说，Go语言主要关注服务器领域的开发，但这不会是Go语言的完整使命。

我们说Go语言适合服务端开发，仅仅是因为它的标准库支持方面，目前是向服务端开发倾斜：

- 网络库（包括socket、http、rpc等）；
- 编码库（包括json、xml、gob等）；
- 加密库（各种加密算法、摘要算法，极其全面）；
- Web（包括template、html支持）。

而作为桌面开发的常规组件：GDI和UI系统与事件处理，基本没有涉及。

尽管Go还很年轻，Go语言1.0版本在2012年3月底发布，到现在才近1年，然而Go语言已经得到了非常普遍的认同。在国外，有人甚至提出“Go语言将制霸云计算领域”。在国内，几乎所有你听到过名字的大公司（腾讯、阿里巴巴、京东、360、网易、新浪、金山、豆瓣等），都有团队对Go语言做服务端开发进行了小范围的实践。这是不能不说是一个奇迹。

Go语言是一门前途非常光明的语言，很少有语言在如此年轻的时候就得到如此热捧。

但因为年轻，导致了Go语言的书籍哪怕在全球都非常稀少。这本书由知名技术作家Mark Summerfield撰写，它会让你了解Go语言，按

Go语言的方式思考，以及使用Go语言来编写高性能软件。一直以来，Summerfield的教学方式都是深入实践的。每一章节都提供了多个活生生的代码示例，它们都是经过精心设计的用于鼓励读者动手实验并且能够帮助读者快速掌握如何开发的。

许式伟

2013年6月

致谢

我写每一本技术书时都得到过来自他人的帮助与建议，本书也不例外。

我想特别感谢两个之前没有Go语言编程经验的程序员朋友——asmin Blanchette和Trenton Schulz。他们两个曾多年为我的书贡献诸多。他们对本书的反馈也让本书能更符合程序员初学Go语言时的需求。

来自Go语言核心开发者Nigel Tao的反馈也让本书受益良多。虽然我并未完全采纳他的所有建议，但是他的反馈总是能够提点我，进而给代码以及书的内容带来极大的改进。

此外，我得到过其他许多人的帮助，包括Go语言初学者David Boddie。他提供了一些有价值的反馈。同时，Go语言的开发者Ian Lance Taylor特别是Russ Cox为我解决了很多代码以及概念上的问题，他们提供的清晰准确的解释对本书的精确性有极大的贡献。

在撰写本书时，我在golang-nuts这个邮件列表里提了许多问题，每次提问总能从众多回邮件者那里收到深思熟虑且实用的回复。同时，Safari上的本书初稿读者也给了我许多反馈，从而让本书中的一些讲解清晰了很多。

意大利的软件公司 www.develer.com 以 Giovanni Bajo 个人的名义，给我提供免费的Mercurial代码库托管服务，让我在写作的漫长过程中能够静心思考。谢谢Lorenzo Mancini为我设置整个环境然后帮我

打理它。同时，我也非常感谢Anton Bowers以及Ben Thompson，自2011年初起，我的网站www.qtrac.eu就托管在他们的网络服务器上。

谢谢Russel Winder在他的博客www.russel.org.uk上讨论软件专利的事情，附件B中有许多思想是从他那里来的。

然后，我要一如既往地感谢lout排版系统的作者Jeff Kingston，我所有的书以及许多其他写作项目都是用这个系统排版而成的。

特别感谢我的责任编辑Debra Willians Cauley，是他将本书成功带给出版社，同时也在本书的写作过程中提供了支持与实际帮助。

同时也感谢出版经理 Anna Popick，他再次将书的出版过程管理得如此好，也感谢校对人员Audrey Doyle的出色工作。

与以往一样，我还要感谢我的妻子Andrea，谢谢她的爱与支持。

引言

本书介绍如何使用Go语言的语言特性以及标准库中的常用包来进行地道的Go语言编程。同时，本书也设计成在学会 Go语言后依然有用的参考资料。为了实现这两个目标，这本书覆盖面非常广，尽量保证每一章只涵盖一个主题，各章之间会进行内容上的交叉引用。

从语言的设计精神来说，Go语言与 C 语言非常相似，是一门精小而高效的语言，它有便利的底层设施，如指针。不过Go语言还提供了许多只在高级或者非常高级的语言中才有的特性，如 Unicode 字符串、强大的内置数据结构、鸭子类型、垃圾收集和高层次的并发支持，使用通信而非常规的共享数据和锁方式。另外，Go语言还提供了一个庞大且覆盖面全的标准库。

虽然所有的Go语言特性或者编程范式都会以完整可运行的示例来详细讲解，但是本书还是假设读者有主流编程语言的经验，比如C、C++、Java、Python或其他类似的语言。

要学好任何一门语言，使用它进行编程都是必经之路。为此，本书采用完全面向实战的方式，鼓励读者亲自去练习书中的例子，尝试着去解决练习题中给出的问题，自己去写程序，以获得宝贵的实践经验。正如我以前写的书一样，本书中所引用的代码片段都是“活代码”。也就是说，这些代码自动提取自.go源文件，并直接嵌入到提供给出版商的PDF文件中，故此不会有剪切和粘贴错误，可以直接运行。只要有可能，本书都会提供小而全的程序或者包来作为贴近实际

应用场景的例子。本书的例子、练习和解决方案都可以从www.qtrac.eu/gobook.html这个网址获得。

本书的主要目的是传授Go语言本身，虽然我们使用了Go语言标准库中的许多包，但不会试图全都涉及。这并不是问题，因为本书向读者提供了足够的Go语言知识来使用任何标准库中的包或者是任何第三方Go语言的包，当然还能够创建自己的包。

为什么是Go

Go语言始于2007年，当时只是Google内部的一个项目，其最初设计者是Robert Griesemer、Unix泰斗Rob Pike和Ken Thompson。2009年11月10日，Go语言以一个自由的开源许可方式公开亮相。Go语言由其原始设计者加上Russ Cox、Andrew Gerrand、Ian Lance Taylor以及其他许多人在内的一个Google团队开发。Go语言采取一种开放的开发模式，吸引了许多来自世界各地的开发者为这门语言的发展贡献力量。其中有些开发者获得了非常好的声望，因此他们也获得了与Google员工一样的代码提交权限。此外，Go Dashboard 这个网站（godashboard.appspot.com/project）也提供了许多第三方的Go语言包。

Go语言是近15年来出现的最令人兴奋的新主流语言。它是第一个直接面向21世纪计算机和开发者的语言。

Go语言被设计为可高效地伸缩以便构建非常大的应用，并可在普通计算机上用几秒钟即完成编译。快如闪电的编译速度可能在一定程度上是因为语言的语法很容易解析，但更主要是因为它的依赖管理。如果文件app.go依赖于文件pkg1.go，而pkg1.go又依赖于pkg2.go，在传统的编译型语言中，app.go需要依赖于pkg1.go和pkg2.go目标文件。但在Go语言中，一切pkg2.go导出的内容都被缓存在pkg1.go的目标文件中，所以pkg1.go的目标文件足够独立构建 app.go。对于只有三个源文

件的程序来说，这看不出什么优劣，但对于有着大量依赖关系的大型应用程序来说，这样做可以获得巨大的编译速度提升。

由于Go语言程序的构建是如此之快，因此它也适用一些本来应该使用脚本语言的场景（见“Go语言版Shebang脚本”，参见1.2节）。此外，Go语言可用于构建基于Google App Engine的Web应用程序。

Go语言使用了一种非常干净且易于理解的语法，避免了像老的语言如C++（发布于1983年）或Java（发布于1995年）一样的复杂和冗长。Go语言是一种强静态类型的语言，这在有些程序员看来是构建大型应用程序的必备特性。然而，使用Go语言进行编程并不需要像使用别的静态语言那样打太多的字，这要归功于Go语言简短的“声明并初始化”的变量声明语法（由于编译器会推断类型，因此并不需要显式地写明），以及它对鸭子类型强大而便捷的支持。

像C和C++这样的语言，当涉及内存管理时需要程序员非常谨慎地面对，特别是对于并发程序，要跟踪它们的内存分配简直犹如噩梦，而这些本来可以交给计算机去做。近年来，C++在这方面用各种“智能”指针进行了很大的改善，但在线程库方面还一直在追赶Java。通过使用垃圾收集器，Java减轻了程序员管理内存的负担。虽然C++语言现在有一个标准的线程库，但是C语言还只能使用第三方线程库。然而，在C、C++或Java中编写并发程序仍然需要相当地谨慎，以确保在恰当的时间正确地锁定和解锁资源。

Go编译器和运行时系统会处理这些繁琐的跟踪问题。对于内存管理而言，Go语言提供了一个垃圾收集器，因此无需使用智能指针或者手动释放内存。Go语言的并发机制基于计算机科学家C.A.R.Hoare提出的CSP（Communicating Sequential Processes）模型构建，这意味着许多并发的Go语言程序不需要加任何锁 [1]。此外，Go语言引入goroutine——一种非常轻量级的进程，可以一次性大量创建，并可跨处理器和处理器核心自动进行负载平衡，以提供比老的基于线程的语

言更细粒度的并发。事实上，因为Go语言的并发支持使用起来如此简单和自然，移植单线程程序到Go时经常会发现转为并发模型的机会大增，从而可以更充分地利用计算机资源。

Go语言是一门务实的语言，与语言的纯净度相比，它更关注语言效率以及为程序员带来的便捷性。例如，Go语言的内置类型和用户自定义的类型是不一样的，因为前者可以高度优化，后者却不能。Go语言也提供了两个基本的内置集合类型：切片（`slice`，它的实际用途是为了提供变长功能的数组）和映射（`map`，也叫键值字典或散列表）。这些集合类型非常高效，并且在大多数情况下都能非常好地满足需求。当然，Go语言也支持指针（它是一个完全编译型的语言，因此在性能方面没有虚拟机挡路），所以它可以轻松创建复杂的自定义类型，如平衡二叉树。

虽然C语言仅支持过程式编程，而Java则强制要求程序员按照面向对象的方式来编程，但Go语言允许程序员使用最合适的编程范式来解决问题。Go语言可以被用做一个纯粹的过程式编程语言，但对面向对象编程也支持得很好。不过，我们也会在后文看到，Go语言面向对象编程的方式与C++、Java或Python非常不同，它更容易使用且在形式上更加灵活。

就像C语言一样，Go语言也不支持泛型（用C++的话来说就是模板）。然而，Go语言所提供的别的功能特性消除了对泛型支持的依赖。Go并不使用预处理器或者包含文件（这也是为什么它编译得如此之快的另一个原因），因此也无需像C和C++那样复制函数签名。同时，因为没有使用预处理器，程序的语义就无法在Go语言程序员背后悄悄变化，但这种情况在C和C++下使用`#define`时一不小心就会发生。

可以说，C++、Objective-C和Java都试图成为更好的C语言（后者是间接地成为了更好的C++语言）。尽管Go语言干净而轻盈的语法容易让人联想到Python，Go语言的切片和映射也非常类似于Python的列

表和字典，但Go语言也可以被认为试图成为一个更好的C。然而，与任何其他语言相比，Go语言从语言本质上都更接近于C语言，并可以被认为保留了C语言的所有精华的同时试图消除C语言中的缺陷，同时加入了许多强大而有用的独有特性。

Go语言最初被构思为一门可充分利用分布式系统以及多核联网计算机优势且适用于开发大型项目的编译速度很快的系统级语言。现在，Go语言的触角已经远远超出了原定的范畴，它正被用做一个具有高度生产力的通用编程语言。使用Go语言开发和维护系统都让人感觉是一种享受。

本书的结构

第1章开始讲解如何建立和运行Go程序。这一章通过5个简短的示例简要介绍了Go语言的语法与特性，以及一些标准库。每个例子都介绍了一些不同的特性。这一章主要是为了让读者尝试一下Go语言，以此让读者感受一下学习Go语言需要学习的大致内容是什么。（这一章还讲解了如何获取和安装Go语言环境。）

第2章至第7章更深入地讲解了Go语言的方方面面。其中有三章专门讲解了Go语言的内置数据类型：第2章涵盖了标识符、布尔值和数值类型，第3章涵盖了字符串，第4章涵盖了Go语言内置的集合类型。

第5章描述并讲解了Go语言的语句和控制结构，还解释了如何创建和使用自定义的函数，最后展示了如何使用Go语言创建一个过程式的非并发程序。

第6章展示了如何在Go语言中进行面向对象编程。本章的内容包括可用于聚合和嵌入（委托）其他类型的结构体，可作为一个抽象类型的接口，以及如何在某些情况下产生类似继承的效果。由于Go语言中进行面向对象编程的方式可能与大多数读者的经验不同，这一章会给出几个完整的例子并详细讲解，以确保读者完全理解Go语言的面向对象编程方式。

第7章讲解了Go语言的并发特性，与面向对象编程一章相比，这一章给出了更多实例，以确保读者对这些新的Go语言特性有透彻的了解。

第8章展示了如何读取和写入自定义的二进制文件、Go二进制（`gob`）文件、文本、JSON以及XML文件。（读取和写入文本文件的知识在第1章和后续几章中都有所涉及，因为这些知识可以更易于提供一些有价值的示例和练习。）

本书的最后一章是第9章。这一章先展示了如何导入和使用标准库包、自定义包以及第三方软件包。它还展示如何对自定义的包进行文档的自动提取、单元测试和性能基准测试。这一章的最后一节对Go编译器（`gc`）提供的工具集以及Go语言的标准库做了简要的概述。

Go语言虽然小巧，但它同时也是一门功能丰富和强大表达能力（在语法结构、概念和编程习惯方面）的语言。本书的例子都符合良好的Go语言编程范式 [2]。当然，这种做法也意味着有些概念出现时不会被当场解释。但我们希望读者相信，所有的概念都会在这本书中进行解释（当然，没有当场解释的内容都会以交叉引用的形式给出相应讲解的位置）。

Go是一门迷人的语言，使用起来感觉非常好。学习Go语法和编程习惯并不会很难，但它的确引入了一些新颖的、对许多读者来说可能不那么熟悉的概念。这本书试图给读者概念上的突破，尤其是在面向对象的Go语言编程和并发Go语言编程方面。如果只阅读那些定义良好却非常简要的文档，读者可能需要花费数周甚至数月的时间才能真正理解相关的知识。

[1].指不需要用户主动加锁，而不是指从内部实现来说没有锁。——译者注

[2].这里有一个例外：前面几章中，即使通道只被当做单向通道使用，我们也总是将通道声明为双向的。从第7章开始，通道只被声明为只有某一特殊的方向，这样，这里所说的Go语言风格用法也就讲得通了。

第1章 5个例子

本章总共有5个比较小的示例程序。这些示例程序概览了Go语言的一些关键特性和核心包（在其他语言里也叫模块或者库，在Go语言里叫做包（`package`），这些官方提供的包统称为Go语言标准库），从而让读者对学习Go语言编程有一个初步的认识。如果有些语法或者专业术语没法立即理解，不用担心，本章所有提到的知识点在后面的章节中都有详细的描述。

要使用Go语言写出Go味道的程序需要一定的时间和实践。如果你想将C、C++、Java、Python以及其他语言实现的程序移植到Go语言，花些时间学习Go语言特别是面向对象和并发编程的知识将会让你事半功倍。而如果你想使用Go语言来从头创建新的应用，那就更要好好掌握Go语言提供的功能了，所以说前期投入足够的学习时间非常重要，前期付出的越多，后期节省的时间也将越多。

1.1 开始

为了尽可能获得最佳的运行性能，Go语言被设计成一门静态编译型的语言，而不是动态解释型的。Go语言的编译速度非常快，明显要快过其他同类的语言，比如C和C++。

Go语言的官方编译器被称为gc，包括编译工具5g、6g和8g，链接工具5l、6l和8l，以及文档查看工具godoc（在Windows下分别是5g.exe、6l.exe等）。这些古怪的命名习惯源自于Plan 9操作系统，例如用数字来表示处理器的架构（5代表ARM，6代表包括Intel 64位处理器在内的AMD64架构，而8则代表Intel 386）。幸好，我们不必担心如何挑选这些工具，因为Go语言提供了名字为go的高级构建工具，会帮我们处理编译和链接的事情。

Go语言官方文档

Go语言的官方网站是golang.org，包含了最新的Go语言文档。其中Packages链接对Go标准库里的包做了详细的介绍，还提供了所有包的源码，在文档不足的情况下是非常有用的。Commands页面介绍了Go语言的命令程序，包括Go编译器和构建工具等。Specification链接主要非正式、全面地描述了Go语言的语法规格。最后，Effective Go链接包含了大量Go语言的最佳实践。

Go语言官网还特地为读者准备了一个沙盒，你可以在这个沙盒中在线编写、编译以及运行Go小程序（有一些功能限制）。这个沙盒对于初学者而言非常有用，可以用来熟悉Go语法的某些特殊之处，甚至可以用来学习fmt包中复杂的文本格式化功能或者regexp包中的正则表达式引擎等。官网的搜索功能只搜索官方文档。如果需要更多其他的Go语言资源，你可以访问go-lang.cat-v.org/go-search。

读者也可以在本地直接查看Go语言官方文档。要在本地查看，读者需要运行godoc工具，运行时需要提供一个参数以使godoc运行为Web服务器。下面演示了如何在一个Unix终端（xterm、gnome-terminal、onsole、Terminal.app或者类似的程序）中运行

```
$ godoc -http=:8000
```

或者在Windows的终端中（也就是命令提示符或MS-DOS的命令窗口）：

```
C:\>godoc -http=:8000
```

其中端口号可任意指定，只要不跟已经运行的服务器端口号冲突就行。假设 `godoc` 命令的执行路径已经包含在你的 `PATH` 环境变量中。

运行 `godoc` 后，你只需用浏览器打开 `http://localhost:8000` 即可在本地查看 Go 语言官方文档。你会发现本地的文档看起来跟 `golang.org` 的首页非常相似。`Packages` 链接会显示 Go 语言的官方标准库和所有安装在 `GOROOT` 下的第三方包的文档。如果 `GOPATH` 变量已经定义（指向某些本地程序和包的路径），`Packages` 链接旁边会出现另一个链接。你可以通过这个链接访问相应的文档（环境变量 `GOROOT` 和 `GOPATH` 将在本章后面小节和第 9 章中讨论）。

读者也可以在终端中使用 `godoc` 命令来查看整个包或者包中某个特定功能的文档。例如，在终端中执行 `godoc image.NewRGBA` 命令将会输出关于函数 `image.NewRGBA()` 的文档。执行 `godoc image/png` 命令会输出关于整个 `image/png` 包的文档。

本书中的所有示例（可以从 www.qtrac.eu/gobook.html 获得）已经在 Linux、Mac OS X 和 Windows 平台上用 Go 1 中的 `gc` 编译器测试通过。Go 语言的开发团队会让所有后续的 Go 1.x 版本都向后兼容 Go 1，因此本书所述文字及示例都适用于整个 1.x 系列的 Go。（如果发生不兼容的情况，我们也会及时更新书中的示例以与最新的 Go 语言发布版兼容。因此，随着时间的推移，网站上的示例程序可能跟本书中所展示的代码不完全相同。）

要下载和安装 Go，请访问 golang.org/doc/install.html，那里有安装指南和下载链接。在撰写本书时，Go 1 已经发布了适用于 FreeBSD 7+、Linux 2.6+、Mac OS X（Snow Leopard 和 Lion）以及 Windows 2000+ 平台的源代码和二进制版本，并且同时支持这些平台的 Intel 32 位和 AMD 64 位处理器架构。另外 Go 1 还在 Linux 平台上支持 ARM 架构。预编译的 Go 安装包已经包含在 Ubuntu Linux 的发行版中，而在你

阅读本书时可能更多的其他Linux发行版也包含Go安装包。如果只为了学习Go语言编程，从Go安装包安装要比从头编译和安装Go环境简单得多。

用gc构建的程序使用一种特定的调用约定。这意味着用gc构建的程序只能链接到使用相同调用约定的外部包，除非出现合适的桥接工具。Go语言支持在程序中以cgo工具（golang.org/cmd/cgo）的形式调用外部的C语言代码。而且目前至少在Linux和BSD系统中已经可以通过SWIG工具（www.swig.org）在Go程序中调用C和C++语言的代码。

除了gc之外还有一个名为gccgo的Go编译器。这是一个针对Go语言的gcc（GNU编译工具集）前端工具。4.6以上版本的gcc都包含这个工具。像gc一样，gccgo也已经在部分Linux发行版中预装。编译和安装gccgo的指南请查看这个网址：golang.org/doc/gccgo_install.html。

1.2 编辑、编译和运行

Go程序使用UTF-8编码 [1] 的纯Unicode文本编写。大部分现代编辑器都能够自动处理编码，并且某些最流行的编辑器还支持Go语言的语法高亮和自动缩进。如果你用的编辑器不支持Go语言，可以在Go语言官网的搜索框中输入编辑器的名字，看看是否有合适的插件可用。为了编辑方便，所有的Go语言关键字和操作符都使用ASCII编码字符，但是Go语言中的标识符可以是任一Unicode编码字符后跟若干Unicode字符或数字，这样Go语言开发者可以在代码中自由地使用他们的母语。

Go语言版Shebang脚本

因为Go的编译速度非常快，Go程序可以作为类Unix系统上的shebang `#!` 脚本使用。我们需要安装一个合适的工具来实现脚本效果。在撰写本书的时候已经有两个能提供所需功能的工具：`gonow` (github.com/kison/gonow) 和 `gorun` (wiki.ubuntu.com/gorun)

在安装完`gonow`或者`gorun`后，我们就可以通过简单的两个步骤将任意Go程序当做shebang脚本使用。首先，将`#!/usr/bin/env gonow`或者`#!/usr/bin/env gorun`添加到包含`main()`函数（在`main`包里）的`.go`文件开始处。然后，将文件设置成可执行（如用`chmod +x`命令）。这些文件只能够用`gonow`或者`gorun`来编译，而不能用普通的编译方式来编译，因为文件中的`#!`在Go语言中是非法的。

当`gonow`或者`gorun`首次执行一个`.go`文件时，它会编译该文件（当然，非常快），然后运行。在随后的使用过程中，只有当这个`.go`文件自上次编译后又被修改过后才会被再次编译这使得用Go语言来快速而方便地创建各种实用工具成为可能，比如创建系统管理任务。

为了感受一下如何编辑、编译和运行Go程序，我将从经典的“Hello World”程序开始（虽然我们会将其设计得稍微复杂些）。我们首先讨论编译与运行，然后在下一节中详细解读文件`hello/hello.go`中的源代码，因为它包含了一些Go语言的基本思想和特性。

我们可以从www.qtrac.eu/gobook.html得到本书中的所有源码，源代码包解压后将是一个`goeg`文件夹。所以如果我们在`$HOME`文件夹下解压缩，源文件`hello.go`的路径将会是`$HOME/goeg/src/hello/hello.go`。如无特别说明，我们在提到程序的源文件路径时将默认忽略`$HOME/goeg/src`部分，比如在这个例子里 `hello` 程序的源文件路径被描述为`hello/hello.go`（当然，Windows用户必须将“/”替换成“\”，同时使用它们自己解压的路径，如`C:\goeg`或者`%HOME-PATH%\goeg`等）。

如果你直接从预编译Go安装包安装，或从源码编译并以root或Administrator的身份安装，那么你的系统中应该至少有一个环境变量GOROOT，它包含了Go安装目录的路径，同时你系统中的环境变量PATH现在应该已经包含\$GOROOT/bin或%GOROOT%\bin。要查看Go是否安装正确，在终端（xterm、gnome-terminal、konsole、Terminal.app或者类似的工具）里键入以下命令即可：

```
$ go version
```

或者在Windows系统的MS-DOS命令提示符窗口里键入：

```
C:\>go version
```

如果返回的是“command not found”或者“go is not recognized...”这样的错误信息，意味着Go不在环境变量PATH中。如果你用的是类Unix系统（包括Mac OS X），有一个很简单的解决办法，就是将该环境变量加入.bashrc（或者其他shell程序的类似文件）中。例如，作者的.bashrc文件包含这几行：

```
export GOROOT=$HOME/opt/go
```

```
export PATH=$PATH:$GOROOT/bin
```

通常情况下，你必须调整这些值来匹配你自己的系统（当然这只有在go version命令返回失败时才需要这样做）。

如果你用的是Windows系统，可以写一个批处理文件来设置Go语言的环境变量，每次打开命令提示符窗口执行Go命令时先运行这个批处理文件即可。不过最好还是在控制面板里设置Go语言的环境变量，一劳永逸。步骤如下，依次点击“开始菜单”（那个Windows图标）、“控制面板”、“系统和安全”、“系统”、“高级系统设置”，在系统属性对话框中点击“环境变量”按钮，然后点击“新建...”按钮，在其中加入一

个以GOROOT命名的变量以及一个适当的值，如C:\Go。在相同的对话框中，编辑PATH环境变量，并在尾部加入文字；C:\Go\bin——文字开头的分号至关重要！在以上两者中，用你系统上实际安装的Go路径来替代C:\Go，如果你实际安装的Go路径不是C:\Go的话。（再次声明，只有在go version命令返回失败时才需要这样做。）

现在我们假设Go在你机器上安装正确，并且Go bin目录包含PATH中所有的Go构建工具。（为了让新设置生效，可能有必要重新打开一个终端或命令行窗口。）

构建Go程序，有两步是必须的：编译和链接。[\[2\]](#) 所有这两步都由go构建工具处理。go构建工具不仅可以构建本地程序和本地包，并且可以抓取、构建和安装第三方程序和第三方包。

让go的构建工具能够构建本地程序和本地包需满足三个条件。首先，Go的bin目录（\$GOROOT/bin或者 %GOROOT%\bin）必须在环境变量中。其次，必须有一个包含src目录的目录树，其中包含了本地程序和本地包的源代码。例如，本书的示例代码被解压到goeg/src/hello和goeg/src/bigdigits等目录。最后，src目录的上一级目录必须在环境变量GOPATH中。例如，为了使用go的构建工具构建本书的hello示例程序，我们必须这样做：

```
$ export GOPATH=$HOME/goeg
$ cd $GOPATH/src/hello
$ go build
```

相应地，在Windows上也可以这样做：

```
C:\>set GOPATH=C:\goeg
C:\>cd %gopath%\src\hello
C:\goeg\src\hello>go build
```

以上两种情况都假设PATH环境变量中已经包含\$GOROOT/bin或者%GOROOT%\bin。在go构建工具构建好了程序后，我们就可以尝试

运行它。可执行文件的默认文件名跟它所位于的目录名称一致（例如，在类Unix系统中是**hello**，在Windows系统中是**hello.exe**），一旦构建完成，我们就可以运行这个程序了。

```
$/hello
```

```
Hello World!
```

或者

```
$/hello Go Programmers!
```

```
Hello Go Programmers!
```

在Windows上也类似：

```
C:\goeg\src\hello>hello Windows Go Programmers !
```

```
Hello Windows Go Programmers!
```

我们用加粗代码字体的形式显示需要你在终端输入的文字，并以罗马字体的形式显示终端的输出。我们也假设命令提示符是**\$**，但其实是什么都没关系（如Windows下的**C:\>**）。

有一点可以注意到的是，我们无需编译或者显式链接任何其他的包（即使我们将看到**hello.go**使用了3个标准库中的包）。这是为什么Go程序构建得如此快的原因。

如果我们有好几个Go程序，如果它们的可执行程序都可以保存在同一个目录下，由于我们可以一次性将这个目录加入到**PATH**中，这将会非常的方便。幸运的是，**go**构建工具可以用以下方式来支持这样的特性：

```
$ export GOPATH=$HOME/goeg
```

```
$ cd $GOPATH/src/hello
```

```
$ go install
```

同样地，我们可以在Windows上这样做：

```
C:\>set GOPATH=C:\goeg
```

```
C:\>cd %gopath%\src\hello
```

```
C:\goeg\src\hello>go install
```

`go install` 命令跟 `go build` 所做的工作是一样的，唯一不同的是，它将可执行文件放入一个标准路径中（`$GOPATH/bin` 或者 `%GOPATH%\bin`）。这意味着，只需在 `PATH` 中加上一个统一路径（`$GOPATH/bin` 或者 `%GOPATH%\bin`），我们所安装的所有 Go 程序都会包含在 `PATH` 中从而可以在任一路径下直接运行。

除了本书中的示例程序之外，我们可能会想在自己的一个目录下开发自己的 Go 程序和包。要达到这个目的，我们可以将 `GOPATH` 环境变量设置成两个或者多个以冒号分隔的路径（在 Windows 中是以分号分隔）。例如，`export GOPATH=$HOME/app/go:$HOME/goeg` 或者 `SET GOPATH=C:\app\go;C:\goeg`。[\[3\]](#) 在这个情况下我们必须将所有的程序和包的源代码都放入 `$HOME/app/go/src` 或者 `C:\app\go\src` 中。因此，如果我们开发了一个叫 `myapp` 的程序，它的 `.go` 源文件将位于 `$HOME/app/go/src/myapp` 或者 `C:\app\go\src\myapp`。如果我们使用 `go install` 在一个 `GOPATH` 路径下构建程序，而且 `GOPATH` 环境变量包含了两个或者更多个路径，那么可执行文件将被放入相对应源代码目录的 `bin` 文件夹中。

通常，每次构建 Go 程序时 `export` 或者设置 `GOPATH` 环境变量可能很费劲，因此最好是永久性地设置好这个环境变量。前面我们已经提到过，类 Unix 系统可修改 `.bashrc` 文件（或类似的文件）以设置 `GOPATH` 环境变量（参见本书示例中的 `gopath.sh` 文件），Windows 上可通过编写一个批处理文件（参见本书示例中的 `gopath.bat` 文件）或添加 `GOPATH` 到系统的环境变量：依次点击“开始菜单”（那个 Windows 图标）、“控制面板”、“系统和安全”、“系统”、“高级系统设置”，在系统属性对话框中点击“环境变量”按钮，然后点击“新建...”按钮，在其中加入一个以 `GOPATH` 命名的变量以及一个适当的值，如 `C:\goeg` 或 `C:\app\go;C:\goeg`。

虽然Go语言的推荐构建工具是go命令行工具，我们完全可以使用make或者其他现代构建工具，或者使用别的针对Go语言的构建工具，或者给流行集成开发环境如Eclipse和Visual Studio安装合适的插件来进行Go工程的构建。

1.3 Hello Who?

现在我们已经知道怎么编译一个 hello 程序，让我们看看它的代码。不要担心细节，本章所提及的一切（以及更多的内容）在后面的章节中都有详细描述。下面是完整的hello程序（在文件hello/hello.go中）：

```
// hello.go
package main
import (①
    "fmt "
    "os "
    "strings "
)
func main() {
    who := " World! " ②
    if len(os.Args) > 1 { /* os.Args[0]是 " hello " 或者 " hello.exe " */
③
        who = strings.Join(os.Args[1:], " ") ④
    }
    fmt.Println( " Hello " , who) ⑤
```

```
}
```

Go语言使用 C++风格的注释：//表示单行注释，到行尾结束，/.../表示多行注释。Go语言中的惯例是使用单行注释，而多行注释则往往用于在开发过程中注释掉若干行代码。[4]

所有的Go语言代码都只能放置于一个包中，每一个Go程序都必须包含一个main包以及一个 main()函数。main()函数作为整个程序的入口，在程序运行时最先被执行。实际上，Go语言中的包还可能包含init()函数，它先于main()函数被执行，我们将在1.7节了解到，关于init函数的完全介绍在5.6.2节。需要注意的是，包名和函数名之间不会发生命名冲突情况。

Go语言针对的处理单元是包而非文件，这意味着我们可以将包拆分成任意数量的文件。在Go编译器看来，如果所有这些文件的包声明都是一样的，那么它们就同样属于一个包，这跟把所有内容放在一个单一的文件里是一样的。通常，我们也可以根据应用程序的功能将其拆分成尽可能多的包，以保持一切模块化，我们将在第9章看到相关内容。

代码中的import语句（标注为①的地方）导入了3个标准库中的包。fmt包提供来格式化文本和读入格式文本的函数（参见 3.5 节），os 包提供了跨平台的操作系统层面变量及函数，而strings包则提供了处理字符串的函数（参见3.6.1节）。

Go语言的基本类型支持常用的操作符（如+操作符可用于数字加法运算和字符串连接运算），同时Go语言的标准库也提供了拥有各种功能的包来对这些操作进行补充，如这里引入的strings包。你也可以基于这些基本类型创建自己的类型或者为这些类型添加自定义方法（我们将在1.5节提及，并在第6章详细阐述）。

读者可能也已经注意到程序中没有分号，那些 import 语句也不用逗号分隔，if 语句的条件也不用圆括号括起来。在Go语言中，包含函

数体以及控制结构体（例如if语句和for循环语句）在内的代码块均使用花括号作为边界符。使用代码缩进仅仅是为了提高代码可读性。从技术层面讲，Go语言的语句是以分号分隔的，但这些是由编译器自动添加的，我们不用手动输入，除非我们需要在同一行中写入多个语句。没有分号及只需要少量的逗号和圆括号，使得Go语言的程序更容易阅读，并且可以大幅降低编写代码时的键盘敲击次数。

Go语言的函数和方法以关键字func定义。但main包里的main()函数比较特别，它既没有参数，也没有返回值。当main.main()运行完毕，程序会自动终止并向操作系统返回0。通常我们可以随时选择退出程序，并返回一个自己选择的返回值，这点我们随后将详细讲解（参见1.4节）。

main()函数中的第一行（标注②）使用了:=操作符，在Go语言中叫做快速变量声明。这条语句同时声明并初始化了一个变量，也就是说我们不必声明一个具体类型的变量，因为Go语言可以从其初始化值中推导出其类型。所以这里我们相当于声明了一个string类型的变量who，而且由于go是强类型的语言，也就只能将string类型的值赋值给who。

就像大多数语言使用if语句检测一个条件是否成立一样，在这个例子里if语句用来判断命令行中是否输入了一个字符串，如果条件成立就执行相应大括号中的代码块。我们将在本章末尾（参见1.6节）及后面的章节（参见5.2.1节）中看到一些更加复杂的if语句。

代码中的os.Args变量是一个string类型的切片（标注③）。数组、切片和其他容器类型将在第4章中详细阐述（参见4.2节）。现在我们只需要知道可以使用语言内置的len()函数来获得切片的长度即可，而切片的元素则可以通过[]索引操作来获得，其语法是一个Python语法子集。具体而言，slice[n]返回切片的第n个元素（从0开始计数），而slice[n:]则返回另一个包含从第n个元素到最后一个元素的切片。在数

据集合那一章节，我们将会看到Go语言在这方面的详细语法。对于 `os.Args`，这个切片总是至少包含一个 `string`（程序本身的名字），其在切片中的位置索引为0（Go语言中的所有索引都是从0开始的）。

只要用户输入一个或多个命令行参数，`if` 语句的条件就成立了，我们将从命令行输入的所有参数连接成一个字符串并赋值给 `who` 变量（标注④）。在这里我们使用赋值操作符（`=`），因为如果我们使用快速声明操作符（`:=`）的话，只能得到另一个生命周期仅限于当前 `if` 代码块的新局部变量 `who`。`strings.Join()` 函数的输入参数为以一个 `string` 类型的切片和一个分隔符（可以是一个空字符，如 `" "`）作为输入，返回一个由分隔符将切片中的所有字符串连接在一起的新字符串。在这个示例里我们用空格作为连接符来连接所有输入的字符串参数。

最后，在最后一个语句（标注⑤）中，我们打印 `Hello` 和一个空格，以及 `who` 变量中的字符串，并添加一个换行符。`fmt` 包提供了许多不同的打印函数变体，比如像 `fmt.Println()` 会整洁地打印任何输入的内容，而像 `fmt.Printf()` 则使用占位符来提供良好的格式化输出控制能力。打印函数将在第3章（参见3.5节）详细阐述。

本节的 `hello` 程序展示了很多超出这类程序一般所做事情之外的语言特性。接下来的示例也会这样做，在保持程序尽量简短的情况下尽量覆盖更多的高级特性。这样做的主要目的是，通过熟悉简单的语言基础，让读者在构建、运行和体验简单的Go程序的同时体验一下Go语言的强大与独特。当然，本章提及的所有内容都将在后面章节中更详细地阐述。

1.4 大数字——二维切片

示例程序**bigdigits**（源文件是**bigdigits/bigdigits.go**）从命令行接收一个数字（作为一个字符串输入），然后用大数字的格式将这个数字输出到命令行窗口。回溯到20世纪，在一些多个用户共用一台高速行式打印机的地方，通常都会习惯性地为每个用户的打印任务添加一个封面页以显示该用户的一些标识信息，比如他们的用户名和打印的文件名等。那时候采取的就是类似于这个例子中演示的大数字技术。

我们将分3部分了解这个示例程序：首先介绍**import**部分，然后是静态数据，再之后是程序处理过程。为了让大家对整个过程有个大致的印象，我们先来看看程序的运行结果，如下：

```
$.bigdigits 290175493
  222    9999    000    1  77777  55555    4    9999
333
  2  2  9    9  0  0  11    7 5    44  9  9  3  3
    2  9    9  0    0  1    7  5    4 4  9  9
  3
    2    9999  0    0  1    7    555  4  4    9999
33
  2    9  0    0  1    7    5  444444    9
3
  2    9  0  0    1  7    5  5    4    9  3  3
22222    9    000    111 7    555    4    9    333
```

从这个例子可以看出，每个数字都由一个字符串类型的切片来表示，所有的数字可以用一个二维的字符串类型切片来表示。在查看数据之前，我们先来了解如何声明和初始化一维的字符串类型以及数字类型的切片。

```
longWeekend := []string{ " Friday " , " Saturday " , " Sunday " , "
Monday " }
```

```
var lowPrimes = []int{2, 3, 5, 7, 11, 13, 17, 19}
```

切片的表达方式为[]Type，如果我们希望同时完成初始化的话，可以在后面直接跟一个花括号，括号内是一个对应类型的元素列表，并在元素之间用逗号分隔。本来对于这两个切片我们可以用同样的变量声明语法，但我们刻意地对 **LowPrimes** 切片的声明采用了相对较长的声明方式。采取这个方式的原因我们很快会给出说明。因为一个切片的类型本身可以是另一个切片，所以我们可以很容易地创建多维的集合（例如元素类型为切片的切片等）。

bigdigits程序只需要引入四个包：

```
import (  
    "fmt "  
    "log "  
    "os "  
    "path/filepath "  
)
```

fmt包提供了格式化文本和读取格式化文本的相关函数（参见3.5节）。**log**包提供了日志功能。**os**包提供的是平台无关的操作系统级别变量和函数，包括用于保存命令行参数的类型为[]string的**os.Args**变量（即字符串类型的切片）。而**path**包中的**filepath**子包则提供了一系列可跨平台的对文件名和路径操作的函数。需要注意的是，对于位于其他包内的子包，在我们的代码中用到时只需要指定其包名称的最后一部分即可（对于此例而言就是**filepath**）。

对于**bigdigits**程序而言，我们需要二维数据（字符串类型的二维切片）。下面我们示范一下如何创建这样的数据，通过将数字0排列好以展示数字对应的字符串如何对应到输出里的行，不过省略了数字3到8的对应字符串。

```
var bigDigits = [][]string{
```

```

{ " 000 ",
  " 0 0 ",
  " 0 0 ",
  " 0 0 ",
  " 0 0 ",
  " 0 0 ",
  " 000 " },
{ " 1 ", " 11 ", " 1 ", " 1 ", " 1 ", " 1 ", " 111 " },
{ " 222 ", " 2 2 ", " 2 ", " 2 ", " 2 ", " 2 ", " 2222 " },
//...3至8...
{ " 9999 ", " 9 9 ", " 9 9 ", " 9999 ", " 9 ", " 9 ", " 9 " },
}

```

虽然在函数和方法之外声明的变量不能使用 `:=` 操作符，但我们可以通过使用关键字 `var` 和赋值运算符 `=` 的长声明方式来达到同样的效果，例如本例中我们为 `bigDigits` 变量所做的。其实之前我们在声明 `lowPrimes` 变量时已经使用过了。不过我们仍然不需要指定 `bigDigits` 的数据类型，因为Go语言能够从赋值动作中推导出相应的类型信息。

我们把计数工作丢给了Go编译器，因此不需要明确指定切片的维度。Go语言的众多便利之一就是支持像大括号这样的复合文面量语法，因此我们不必在一个地方声明这个变量，又在别的地方将相应的值赋值给它，当然，这么做也是可以的。

`main()` 函数总共只有20行代码，从命令行读取输入然后生成输出结果。

```

func main() {
    if len(os.Args) == 1 { ①

```

```

        fmt.Printf( "    usage:  %s  <whole-number>\n    " ,
filepath.Base(os.Args[0]))
    os.Exit(1)
}
stringOfDigits := os.Args[1]
for row := range bigDigits[0] { ②
    line := " "
    for column := range stringOfDigits { ③
        digit := stringOfDigits[column] - '0' ④
        if 0 <= digit && digit <= 9 { ⑤
            line += bigDigits[digit][row] + " " ⑥
        } else {
            log.Fatal( " invalid whole number " )
        }
    }
    fmt.Println(line)
}
}

```

程序先检查启动时是否带有命令行参数。如果没有，则 `len(os.Args)` 的值为1（回忆一下，`os.Args[0]`存放的是程序名字，因此这个切片的长度通常至少为1），然后if条件成立，调用 `fmt.Printf()` 函数打印一条用法信息，`fmt.Printf()`接收%占位符，类似于 C/C++ 中 `printf()` 函数的支持方式，以及Python的%操作符（更详细的用法可参见3.5节）。

`path/filepath`包提供了路径操作函数。比如，`filepath.Base()`函数会返回传入路径的基础名（其实就是文件名）。输出消息后，程序通过

调用`os.Exit`函数退出，返回1给操作系统。在类Unix系统中，程序返回0表示成功，非零值表示用法问题或执行失败。

`filepath.Base()`函数的用法演示了 Go 语言的一个很酷的功能：在导入一个包时，无论这是一个顶级包还是属于其他包（如 `path/filepath`），我们只需要使用包名里的最后一部分来引用它（如 `filepath`）。而且我们还可以在引入包时给这个包分配一个别名以避免名字冲突。本书第9章会详细介绍相关的用法。

假如用户传入了至少一个命令行参数，我们会将第一个命令行参数复制到 `stringOfDigits` 字符串变量中。为了能够将用户输入的数字转换为大数字，我们需要遍历 `bigDigits` 切片中的每一行，也就是说，先生成每个数字的第一行，然后再生成第二行，等等。我们假设所有的 `bigDigits` 切片都包含了同行的行数，因此我们直接使用了第一个切片的行数。Go 语言的 `for` 循环有若干种不同的语法以满足不同的需求；本例标注②和③的地方我们使用了 `for...range` 循环来返回切片中每个元素的索引位置。

行列循环部分的代码可以用如下方式实现：

```
for row := 0; row < len(bigDigits[0]); row++ {  
    line := " "  
    for column := 0; column < len(stringOfDigits); column++ {  
        ...  
    }  
}
```

这是 C、C++、Java 程序员所熟悉的方式，当然 Go 语言也支持 [5]。但是 `for...range` 语法可以实现得更短且更方便（我会在 5.3 节中讨论 Go 语言中 `for` 循环的各种详细用法）。

在每次遍历行之前我们会将行的 `line` 变量设置为一个空字符串。然后我们再遍历从用户那里接受到的 `stringOfDigits` 字符串中的每一列（其实就是字符）。Go 语言中的字符串采用的是 UTF-8 编码，因此一个字符有可能占用两个或者更多字节。不过这在本例中并不是个问题。

题，因为我们只需要考虑如何处理0到9的数字，而这些数字在UTF-8中都是用一个字节表示。它们的表示方法与7位的ASCII标准完全一致。（之后在第3章中我们将学习如何一个字符一个字符地遍历一个字符串，无论其中的字符是单字节还是多字节。）

当我们按索引位置查询一个字符串的内容时，我们将得到索引位置对应的一个byte类型的值（在Go语言中，byte类型等同于uint8类型）。所以，我们可以对命令行传入的参数按索引位置取相应的byte类型值，然后将该值和数字0对应的byte类型值相减，以得知对应的数字。在UTF-8和ASCII中，字符‘0’对应的是48，字符‘1’对应的是49，以此类推。因此，假如我们得到的是一个字符‘3’（对应数值为51），那么我们可以通过运算‘3’-‘0’（也就是51-48）来获取相应的整型值，也就是一个byte类型的整型数，值为3。

Go语言采用单引号来表达字符，而一个字符其实就是一个与Go语言所有其他整型类型兼容的整型数。Go语言的强类型特征意味着我们不能在不做强制类型转换的前提下将一个int32类型和一个int16类型直接相加，但Go语言的数值类型常量适应到它们的上下文，因此在这个上下文里，‘0’将会被当做是一个byte类型。

假如对应的数字在范围之内，我们可以添加合适的字符串到该行中（在if语句中常量0和9被认为是byte类型，因为digit的类型就是byte，但如果digit是其他的一个类型，比如是int，那么它们也自然会被认为是相应的类型）。虽然Go语言的字符串是不可变的，但 += 这种语法在Go语言里也是支持的，主要是易于使用，实质上是暗地里将原字符串替换掉了，另外 + 连接运算符也是支持的，返回一个将两个字符串连接起来的新字符串（第3章将对字符串进行详细描述）。

为了获得对应的字符串，我们先访问对应于数字的bigDigits切片中的相应行。

如果数字超过了范围（比如包含了非数字的字符），我们调用 `log.Fatal()` 函数记录一条错误信息，包括日期、时间和错误信息，如果没有显式指定记录到哪里，那么默认是打印到 `os.Stderr`，并调用 `os.Exit(1)` 终止程序的执行。另外还有一个 `log.FatalF()` 函数可以接受%格式的占位符。在第一个 `if` 语句里我们没有使用 `log.Fatal()` 函数，因为我们只需要输出程序的帮助信息，而不需要日期和时间这些通常 `log.Fatal()` 函数的输出会包含的信息。

当每个数字对应行的字符串准备就绪后，这一行将被打印。在这个例子中，总共有7行被打印，因为每个 `bigDigits` 字符串切片中的数字都用七个字符串来表示。

最后一点，通常情况下声明和定义的顺序并不会带来影响。因此在 `bigdigits/bigdigits.go` 文件中，我们可以在 `main()` 函数前后声明 `bigDigits` 变量。在这个例子中，我们将 `main()` 函数放在前面，因为本书所有的例子我们都趋向于用自上而下的方式来组织内容。

这两个例子中我们已经接触到不少东西，但也仅仅是介绍了 Go 语言与其他主流语言类似的一些功能，除了语法上略有区别外。接下来的3个例子将把我们带离舒适地带，开始展示Go语言的一些特有功能，比如特有的Go语言类型，文件处理（包括错误处理）和以值方式传递函数，以及使用 `goroutine` 和通道（`channel`）进行并行编程等。

1.5 栈——自定义类型及其方法

虽然Go语言支持面向对象编程，但它既没有类也没有继承（`is-a` 关系）这样的概念。但是Go语言支持创建自定义类型，而且很容易创建聚合（`has-a` 关系）结构。Go语言也支持将其数据和行为完全分离，

同时也支持鸭子类型。鸭子类型是一种强有力的抽象机制，它意味着数据的值（比如传入函数的数据）可以根据该数据提供的方法来被处理，而不管其实际的类型。这个术语是从这条语句演化而来的：“如果它走起来像鸭子，叫起来像鸭子，它就是一只鸭子。”所有这些一起，提供了一种游离于类和继承之外的更加灵活强大的选择。但如果要从Go语言的面向对象特性中获益，习惯于传统方法的我们必须在概念上做一些重大调整。

Go语言使用内置的基础类型如 `bool`、`int`和`string` 等类型来表示数据，或者使用`struct`来对基本类型进行聚合。[\[6\]](#) Go语言的自定义类型建立在基本类型、`struct`或者其他自定义类型之上。（我们会在本章后面看到一些简单的例子，参见1.7节。）

Go语言同时支持命名和匿名的自定义类型。相同结构的匿名类型等价，可以相互替换，但是不能有任何方法（这点我们会在6.4节详细阐述）。任何命名的自定义类型都可以有方法，并且这些方法一起构成该类型的接口。命名的自定义类型即使结构完全相同，也不能相互替换（除特别声明之外，本书所指的“自定义类型”都是指命名的自定义类型）。

接口也是一种类型，可以通过指定一组方法的方式定义。接口是抽象的，因此不可以实例化。如果某个具体类型实现了某个接口所有的方法，那么这个类型就被认为实现了该接口。也就是说，这个具体类型的值既可以当做该接口类型的值来使用，也可以当做该具体类型的值来使用。然而，不需要在接口和实现该接口的具体类型之间建立形式上的联接。一个自定义的类型只要实现了某个接口定义的所有方法就是实现了该接口。当然，一个类型可以实现多个接口，只要这个类型同时实现多个接口所定义的所有方法。

空接口（没有定义方法的接口）用`interface{}`来表示。[\[7\]](#) 由于空接口没有做任何要求（因为它不需要任何方法），它可以用来表示任

意值（效果上相当于一个指向任意类型值的指针），无论这个值是一个内置类型的值还是一个自定义类型的值（Go语言的指针和引用将在4.1节介绍）。顺便提一句，在Go语言中我们只讲类型和值，而非类和对象或者实例（因为Go语言没有类的概念）。

函数和方法的参数类型可以是任意内置类型或者自定义类型，甚至是接口。后一种情况表示，一个函数可能接收这样一个参数，例如“传入一个可以读取数据的值”，而不管该值的实际类型是什么（我们马上会在实践中看到这个，参见1.6节）。

第6章详细阐述了这些，并提供了许多例子来保证读者理解这些想法。现在，就让我们来看一个非常简单的自定义栈类型如何被创建和使用，然后看看该自定义类型是如何实现的。

我们从程序的运行结果分析开始：

```
$./stacker
81.52
[pin clip needle]
-15
hay
```

上述结果中的每一项都从该自定义栈中弹出，并各自在单独一行中打印出来。

这个程序的源码是stacker/stacker.go。这里是该程序的包导入语句：

```
import (
    "fmt"
    "stacker/stack"
)
```

fmt包是Go语言标准库的一部分，而stack包则是为我们的stacker程序特意创建的一个本地包。一个Go语言程序或者包的导入语句会首先

搜索GOPATH定义的路径，然后再搜索GOROOT所定义的路径。在这个例子中，程序的源代码位于\$HOME/goeg/src/stacker/stacker.go中，而stack包则位于\$HOME/goeg/src/stacker/stack/stack.go中。只要GOPATH是\$HOME/goeg或包含了\$HOME/goeg这个路径，go构建工具就会将stack和stacker都构建好。

包导入的路径使用Unix风格的“/”来声明，就算在Windows平台上也是这样。每一个本地包都需要保存在一个与包名同名的目录下。本地包可以包含它们自己的子包（如path/filepath），其形式与标准库完全相同（创建和使用自定义包的内容将在第9章中详细阐述）。

下面是打印出输出结果的简单测试程序的main()函数：

```
func main() {
    var haystack stack.Stack
    haystack.Push( " hay " )
    haystack.Push(-15)
    haystack.Push([]string{ " pin " , " clip " , " needle " })
    haystack.Push(81.52)
    for {
        item, err := haystack.Pop()
        if err != nil {
            break
        }
        fmt.Println(item)
    }
}
```

函数的开头声明了一个stack.Stack类型的变量haystack。在Go语言中，导入包中的类型、函数、变量以及其他项的惯例是使用pkg.item这样的语法。其中，pkg是包名中的最后一部分（或唯一一项）。这样有

助于避免名字冲突。然后，我们往栈中压入一些元素，并将其逐一弹出后再输出，直至栈被清空。

使用自定义栈的一个奇妙之处在于可以自由地将异构（类型不同）的元素混合存储，而不仅仅是存储同构（类型相同）的元素。虽然Go语言是强类型的，但是我们可以通过空接口来实现这一点。我们这个例子中的`stack.Stack`类型就是这么做的，无需关心它们的实际类型是什么。当然，在实际使用中，这些元素的实际类型我们还是要知道的。不过，在这里我们只使用到了`fmt.Println()`函数，它可以使用Go语言的类型检视功能（在`reflect`包中）来获得它要打印的元素的类型信息（反射将在后面的9.4.9节中讲到）。

这段代码展示的另一个Go语言的美妙特性就是不带条件的for循环。这是一个无限循环，因此大部分情况下，我们需要提供一种方法来跳出循环，比如这里使用的`break`语句或者一个`return`语句。我们会在下一个例子中看到另一种for循环语法（参见1.6节）。for循环的完整语法将在第5章叙述。

Go语言的函数和方法均可返回单一值或者多个值。Go语言中报告错误的惯例是函数或者方法的最后一个返回值是一个错误值（其类型为`error`）。我们的自定义类型`stack.Stack`也遵从这样的惯例。

既然我们知道自定义类型`stack.Stack`是怎么使用的，就让我们再来看看它的具体实现（源码在文件`staker/stack/stack.go`中）。

```
package stack
import "errors"
type Stack []interface{}
```

按照惯例，该文件开始处声明其包名，然后导入需要使用的包，在这里只有一个包，即`errors`。

在Go语言中定义一个命名的自定义类型时，我们所做的是将一个标识符（类型名称）绑定在一个新类型上，这个新类型与已有的（内

置的或者自定义的) 类型有相同的底层表示。但Go语言又会认为这两个底层表示有所区别。在这里, **Stack**类型只是一个空接口类型切片 (也就是一个可变长数组的引用) 的别名, 但它与普通的`[]interface{}`类型又有所区别。

由于Go语言的所有类型都实现了空接口, 因此任意类型的值都可以存储在**Stack**中。

内置的数据集合类型 (映射和切片)、通信通道 (可缓冲) 和字符串等都可以使用内置的**len()**函数来获取其长度 (或者缓冲大小)。类似地, 切片和通道也可以使用内置的**cap()**函数来获取容量 (它可能比其使用的长度大)。(Go语言的所有内置函数都以交叉引用的形式列在表5-1中, 切片在第4章有详细阐述, 参见4.2节。)通常所有的自定义数据集合类型 (包括我们自己实现的以及Go语言标准库中的自定义数据集合类型) 都应实现**Len()**和**Cap()**方法。

由于 **Stack** 类型使用切片作为其底层表示, 因此我们应为其实现**Stack.Len()**和**Stack.Cap()**方法。

```
func (stack Stack) Len() int {  
    return len(stack)  
}
```

函数和方法都使用关键字**func**定义。但是, 定义方法的时候, 方法所作用的值的类型需写在 **func**关键字之后和方法名之前, 并用圆括号包围起来。函数或方法名之后, 则是小括号包围起来的参数列表 (可能为空), 每个参数使用逗号分隔 (每个参数以**variableName type**这种形式声明)。参数后面, 则是该函数的左大括号 (如果它没有返回值的话), 或者是一个单一的返回值 (例如, **Stack.Len()**方法中的**int**返回值), 也可以是一对圆括号包围起来的返回值列表, 后面再紧跟着一个左大括号。

大部分情况下，会为调用该方法的值命名，例如这里我们使用 `stack` 命名（并且与其包名并不冲突）。调用该方法的值在Go语言中以术语“接收器”来称呼 [8]。

本例中，接收器的类型是 `Stack`，因此接收器是按值传递的。这也意味着任何对该接收器的改变都只是作用于其原始值的一份副本，因此会丢失。这对于不需要修改接收器的方法来说是没问题的，例如本例中的 `Stack.Len()` 方法。

`Stack.Cap()` 方法基本上和 `Stack.Len()` 一样（所以这里没有给出）。唯一的区别是，`Stack.Cap()` 方法返回的是栈的 `cap()` 而非 `len()` 的值。源代码中还包含一个 `Stack.IsEmpty()` 方法，但它也跟 `Stack.Len()` 方法极为相似，只是返回一个 `bool` 值以表示栈的 `len()` 是否等于0，因此也就不再列出。

```
func (stack *Stack) Push(x interface{}) {  
    *stack = append(*stack, x)  
}
```

`Stack.Push()` 方法在一个指向 `Stack` 的指针上被调用（稍后解释），并且接收一个任意类型的值作为参数。内置的 `append()` 函数可以将一个或多个值追加到一个切片里去，并返回一个切片（可能是新建的），该切片包含原始切片的内容和在尾部追加进去的内容。

如果之前有数据从该栈弹出过，则底层的切片容量可能比切片的实际长度大，因此压栈操作会非常的廉价：只需简单地将 `x` 这项保存在 `len(stack)` 这个位置，并将栈的长度加1。

`Stack.Push()` 函数永远有效（除非计算机的内存耗尽），因此我们没必要返回一个 `error` 值来表示成功或者失败。

如果我们要修改接收器，就必须将接收器设为一个指针。[9] 指针是指一个保存了另一个值的内存地址的变量。使用指针的原因之一是为了效率，比如我们有一个很大的值，传入一个指向该值所在内存

地址的指针会比传入该值本身更廉价得多。指针的另外一个用处是使一个值可被修改。例如，当一个变量传入到一个函数中，该函数只得到该值的一份副本（例如，传`stack`给`stack.Len()`函数）。这意味着我们对该值所做的任何改动，对于原始值来说都是无效的。如果我们想修改原始值（就像这里一样我们想往栈中压入数据），我们必须传入一个指向原始值的指针，这样在函数内部我们就可以修改指针所指向的值了。

指针通过在类型名字前面添加一个星号来声明（即星号`*`）。因此，在`Stack.Push()`方法中，变量`stack`的类型为`*Stack`，也就是说变量`stack`保存了一个指向`Stack`类型值的指针，而非一个实际的`Stack`类型值。我们可以通过解引用操作来获取该指针所指向值的实际`Stack`值，解引用操作只是简单意味着我们在试图获得该指针所指处的值。解引用操作通过在变量前面加上一个星号来完成。因此，我们写`stack`时，是指一个指向`Stack`的指针（也就是一个`*Stack`）。写`*stack`时，是指解引用该指针变量，也就是引用该指针所指之处的实际`Stack`类型值。

此外星号处于不同的位置所表达的含义也不尽相同。在两个数字或者变量之间时表示乘法，例如`x*y`，这一点Go和C、C++等是一样的。在类型名称前面时表示指针，例如`*MyType`。在变量名称之前时表示解引用，例如`*Z`。不过不要太担心这些，我们在第4章中将详细阐述Go语言指针的用法。

需要注意的是，Go语言中的通道（`channel`）、映射（`map`）和切片（`slice`）等数据结构必须通过`make()`函数创建，而且`make()`函数返回的是该类型的一个引用。引用的行为和指针非常类似，当把它们传入函数的时候，函数内对该引用所做的任何改变都会作用到该引用所指向的原始数据。然而，引用不需要被解引用，因此大部分情况下不需要将其与星号一起使用。但是，如果我们要在一个函数或者方法内部使用 `append()` 修改一个切片（不同于仅仅修改其中的一个元素内

容），必须要么传入指向这个切片的一个指针，要么就返回该切片（也就是将原始切片设置为该函数或者方法返回的值），因为有时候 `append()` 返回的切片引用与之前所传入的不同。

`Stack` 类型使用一个切片来表示，因此 `Stack` 类型的值也可以在操作切片的函数如 `append()` 和 `len()` 中使用。然而，`Stack` 类型的值仅仅是该类型的值，与其底层表示的类型值不一样，因此如果我们需要修改它就必须传入指针。

```
func(stack Stack) Top() (interface{}, error) {
    if len(stack) == 0 {
        return nil, errors.New( " can't Top en empty stack " )
    }
    return stack[len(stack)-1], nil
}
```

`Stack.Top()` 方法返回栈中最顶层的元素（最后被添加进去的元素）和一个 `error` 类型的错误值，栈不为空时这个错误值为 `nil`，否则不为 `nil`。这个名为 `stack` 的接收器之所以被按值传递，是因为栈没有被修改。

`error` 是一个接口类型（参见6.3节），其中包含了一个方法 `Error() string`。通常，Go语言的库函数的最后一个返回值为 `error` 类型，表示成功（`error` 的值为 `nil`）或者失败。这段代码里我们通过使用 `errors` 包中的 `errors.New()` 函数将 `Stack` 类型设计成与标准库中的类型一样工作。

Go语言使用 `nil` 来表示空指针（以及空引用），即表示指向为空的指针或者引用值为空的引用。[\[10\]](#) 这种指针只在条件判断或者赋值的时候用到，而不应该调用 `nil` 值的成员方法。

Go语言中的构造函数从来不会被显式调用。相反地，Go语言会保证当一个值创建时，它会被初始化成相应的空值。例如，数字默认被初始化成0，字符串默认被初始化成空字符串，指针默认被初始化成

`nil`值，而结构体中的各个字段也被初始化成相应的空值。因此，在Go语言中不存在未初始化的数据，这减少了很多在其他语言中导致出错的麻烦。如果默认初始化的空值不合适，我们可以自己写一个创建函数然后显式地调用它，就像在这里创建一个新的`error`值一样。也可以防止调用者不通过创建函数而直接构造某个类型的值，我们在第6章将详细阐述如何做到这一点。

如果栈不为空，我们返回其最顶端的值和一个`nil`错误值。由于Go语言中的索引从0开始，因此切片或者数组的第一个元素的位置为0，最后一个元素的位置为`len(sliceOrArray) - 1`。

在函数或者方法中返回一个或多个返回值时无需拘泥于形式，只需在所定义函数的函数名后列上返回值类型，并在函数体中保证至少有一个`return`语句能够返回相应的所有返回值即可。

```
func (stack *Stack) Pop() (interface{}, error) {
    theStack := *stack
    if len(theStack) == 0 {
        return nil, errors.New( " Can't pop an empty stack " )
    }
    x := theStack[len(theStack) - 1] ①
    *stack = theStack[:len(theStack) - 1] ②
    return x, nil
}
```

`Stack.Pop()`方法用于删除并返回栈中最顶端（最新添加）的元素。像`Stack.Top()`方法一样，它返回该元素和一个`nil`错误值，或者如果栈为空则返回一个`nil`元素和一个非`nil`错误值。

由于该方法需要通过删除元素来修改栈，因此它的接收器必须是一个指针类型的值。为了方便，我们在方法内不使用`*stack`（`stack`变量实际所指向的栈）这样的语法，而是将其赋值给一个临时变量

(`theStack`)，然后在代码中使用该临时变量。这样做的性能开销非常小，因为 `*stack` 指向的是一个 `Stack` 值，该值使用一个切片来表示，因此这样做的性能开销仅仅比直接使用一个指向切片的引用稍微大一点。

如果栈为空，我们返回一个合适的错误值。否则，我们将该栈最顶端的值保存在一个临时变量 `x` 中，然后对原始栈（本身是一个切片）做一次切片操作（新的切片只是少了一个元素），并将切片后的新栈赋值给 `stack` 指针所指向的原始栈。最后，我们返回弹出的值和一个 `nil` 错误值。`Go` 编译器会重用这个切片，仅仅将其长度减1，并保持其容量不变，而非真地将所有数据拷到另一个新的切片中。

返回的元素通过使用 `[]` 索引操作符和一个索引来得到（标识①）。本例中，该元素索引就是切片最后一个元素的索引。

新的切片通过使用切片操作符 `[]` 和一个索引范围来获得（标识②）。索引范围的形式是 `first:end`。如果 `first` 值像这个示例中一样被省略，则其默认值为0，而如果 `end` 值被省略，则其默认值为该切片的 `len()` 值。新获得的切片包含原切片中从第 `first` 个元素到第 `end` 个元素之间的所有元素，其中包含第 `first` 个元素而不包含第 `end` 个元素。因此，在本例中，通过将其最后一个元素设置为其原切片的长度减1，我们获得了原切片中除最后一个元素外的所有元素组成的切片，快速有效地删除了切片中的最后一个元素（切片索引将在第4章详细阐述，参见4.2.1节）。

对于本例中那些无需修改 `Stack` 的方法，我们将接收器的类型设置为 `Stack` 而非指针（即 `*Stack` 类型）。对于其底层表示较为轻量（比如只包含少量 `int` 类型和 `string` 类型的成员）的自定义类型来说，这是非常合理的。但是对于比较复杂的自定义类型，无论该方法是否需要修改值内容，我们最好一直都使用指针类型的接收器，因为传递一个指针的开销远比传递一个大块的值低得多。

关于指针和方法，有个小细节需要注意的是，如果我们在某个值类型上调用其方法，而该方法所需要的又是一个指针参数，那么Go语言会很智能地将该值的地址（假设该值是可寻址的，参见6.2.1节）传递给该方法，而非该值的一份副本。相应地，如果我们在某个值的指针上调用方法，而该方法所需要的是一个值，Go语言也会很智能地将该指针解引用，并将该指针所指的值传递给方法。[\[11\]](#)

正如本例所示，在Go语言中创建自定义类型通常非常简单明了，无需引入其他语言中的各种笨重的形式。Go语言的面向对象特性将在第6章中详细阐述。

[1.6 americanise示例——文件、映射和闭包](#)

为了满足实际需求，一门编程语言必须提供某些方式来读写外部数据。在前面的小节中，我们概览了Go语言标准库里fmt包中强大的打印函数，本节中我们将介绍Go语言中基本的文件处理功能。接下来我们还会介绍一些更高级的Go语言特性，比如将函数或者方法当做第一类值（**first-class value**）来对待，这样就可以将它们当做参数传递。另外，我们还将用到Go语言的映射（**map**，也称为数据字典或者散列）类型。

本节尽可能详尽地讲述如何编写一个文本文件读写程序，使得示例和相应的练习都更加生动有趣。第8章将会更详尽地讲述Go语言中的文件处理工具。

大约在20世纪中期，美式英语超越英式英语成为最广泛使用的英语形式。本小节中的示例程序将读取一个文本文件，将文本文件中的英式拼写法替换成相应的美式拼写法（当然，该程序对于语义分析和

惯用语分析无能为力），然后将修改结果写入到一个新的文本文件中。这个示例程序的源代码位于**americanise/americanise.go**中。我们采用自上而下的方式来分析这段程序，先讲解导入包，然后是**main()**函数，再到**main()**函数里面所调用的函数，等等。

```
import (  
    "bufio "  
    "fmt "  
    "io "  
    "io/ioutil "  
    "log "  
    "os "  
    "path/filepath "  
    "regexp "  
    "strings "  
)
```

该示例程序所引用的都是 Go 标准库里的包。每个包都可以有任意个子包，就如上面程序中所看到的**io**包中的**ioutil**包以及**path**包中的**filepath**包一样。

bufio包提供了带缓冲的I/O处理功能，包括从UTF-8编码的文本文件中读写字符串的能力。**io**包提供了底层的I/O功能，其中包含了我们的**americanise**程序中所用到的**io.Reader**和**io.Writer**接口。**io/ioutil**包提供了一系列高级文件处理函数。**regexp**包则提供了强大的正则表达式支持。其他的包（**fmt**、**log**、**filepath**和**strings**）已在本书之前介绍过。

```
func main() {  
    inFilename, outFilename, err := filenamesFromCommandLine()①  
    if err != nil {  
        fmt.Println(err) ②
```

```

    os.Exit(1)
}
inFile, outFile := os.Stdin, os.Stdout③
    if inFilename != " " {
        if inFile, err = os.Open(inFilename); err != nil {
            log.Faall(err)
        }
        defer inFile.Close()④
    }
    if outFilename != " " {
        if outFile, err = os.Create(outFilename); err != nil {
            log.Fatal(err)
        }
        defer outFile.Close()⑤
    }
    if err = americanize(inFile, outFile); err != nil {
        log.Fatal(err)
    }
}

```

这个 **main()** 函数从命令行中获取输入和输出的文件名，放到相应的变量中，然后将这些变量传入 **americanise()** 函数，由该函数做相应的处理。

该函数开始时取得所需输入和输出文件的文件名以及一个 **error** 值。如果命令行的解析有误，我们将输出相应的错误信息（其中包含程序的使用帮助），然后立即终止程序。如果某些类型包含 **Error()** **string** 方法或者 **String()** **string** 方法，Go 语言的部分打印函数会使用反射功能来调用相应的函数获取打印信息，否则 Go 语言也会尽量获取能获

取的信息并进行打印。如果我们为自定义类型提供这两个方法中的一个，Go语言的打印函数将会打印该自定义类型的相应信息。我们将在第6章详细阐述相关的做法。

如果err的值为nil，说明变量inFilename和outFilename中包含字符串（可能为空），程序继续。Go语言中的文件类型表示为一个指向os.File值的指针，因此我们创建了两个这样的变量并将其初始化为标准输入输出流（这些流的类型都为*os.File）。正如你在以上程序中所看到的，Go语言的函数和方法支持多返回值，也支持多重赋值操作（标识①和③）。

本质上讲，每一个文件名的处理方式都相同。如果文件名为空，则相应的文件句柄已经被设置成os.Stdin或者os.Stdout（它们的类型都为*os.File，即一个指向os.File类型值的指针），但如果文件名不为空，我们就创建一个新的*os.File指针来读写对应的文件。

os.Open()函数接受一个文件名字符串，并返回一个*os.File类型值，该值可以用来从文件中读取数据。相应地，os.Create()函数接受一个文件名字符串，返回一个*os.File值，该值可以用来从文件中读取数据或者将数据写入文件。如果文件名所指向的文件不存在，我们会先创建该文件，若文件已经存在则会将文件的长度截为0（Go语言也提供了os.OpenFile()函数来打开文件，该函数可以让使用者自由地控制文件的打开模式和权限）。

事实上os.Open()、os.Create()和os.OpenFile()这几个函数都有两个返回值：如果文件打开成功，则返回*os.File和nil错误值；如果文件打开失败，则返回一个nil文件句柄和相应非nil的error值。

返回的err值为nil意味着文件已被成功打开，我们在后面紧跟一个defer语句用于关闭文件。任何属于defer语句所对应的语句（参见5.5节）都保证会被执行（因此需要在函数名后面加上括号），但是该函数只会在defer语句所在的函数返回时被调用。因此，defer语句先“记

住”该函数，并不马上执行。这也意味着`defer`语句本身几乎不用耗时，而执行语句的控制权马上会交给`defer`语句的下一条语句。因此，被推迟执行的`os.File.Close()`语句实际上不会马上被执行，直到包含它的`main()`函数返回（无论是正常返回还是程序崩溃，稍后我们会讨论）。这样，打开的文件就可以被继续使用，并且保证会在我们使用完后自动关闭，即便是程序崩溃了。

如果我们打开文件失败，则调用 `log.Fatal()` 函数并传入相应的错误信息。正如我们在前文中所看的，这个函数会记录日期、时间和相应的错误信息（除非指定了其他输出目标，否则错误记录会默认打印到 `os.Stderr`），并调用 `os.Exit()` 来终止程序。当 `os.Exit()` 函数被直接调用或通过 `log.Fatal()` 间接调用时，程序会立即终止，任何延迟执行的语句都会被丢失。不过这不是个问题，因为 Go 语言的运行时系统会将所有打开的文件关闭，其垃圾回收器会释放程序的内存，而与该程序通信的任何设计良好的数据库或者网络应用都会检测到程序的崩溃，从而从容地应对。正如 `bigdigits` 示例程序中那样，我们不在第一个 `if` 语句（标识②）中使用 `log.Fatal()`，因为 `err` 中包含了程序的使用信息，而且我们不需要打印 `log.Fatal()` 函数通常会输出的日期和时间信息。

在 Go 语言中，`panic` 是一个运行时错误（很像其他语言中的异常，因此本书将 `panic` 直接翻译为“异常”）。我们可以使用内置的 `panic()` 函数来触发一个异常，还可以使用 `recover()` 函数（参见 5.5 节）来在其调用栈上阻止该异常的传播。理论上，Go 语言的 `panic/recover` 功能可以用于多用途的错误处理机制，但我们并不推荐这么用。更合理的错误处理方式是让函数或者方法返回一个 `error` 值作为其最后或者唯一的返回值（如果没错误发生则返回 `nil` 值），并让调用方来检查所收到的错误值。`panic/recover` 机制的目的是用来处理真正的异常（即不可预料的异常）而非常规错误。 [\[12\]](#)

两个文件都成功打开后（`os.Stdin`、`os.Stdout`和`os.Stderr`文件是由Go语言的运行时系统自动打开的），我们将要处理的文件传给`americanise()`函数，由该函数对文件进行处理。如果`americanise()`函数返回`nil`值，`main()`函数将正常终止，所有被延迟的语句（在这里是指关闭`inFile`和`outFile`文件，如果它们不是`os.Stdin`和`os.Stdout`的话）都将被一一执行。如果`err`的值不是`nil`，则错误会被打印出来，程序退出，Go语言的运行时系统会自动将所有打开的文件关闭。

`americanise()`函数的参数是`io.Reader`和`io.Writer`接口，但我们传入的是`*os.File`，原因很简单，因为`os.File`类型实现了`io.ReadWriter`结构（而`io.ReadWriter`是`io.Reader`和`io.Writer`接口的组合），也就是说，`os.File`类型的值可以用于任何要求`io.Reader`或者`io.Writer`接口的地方。这是一个典型的鸭子类型的实例，也就是任何类型只要实现了该接口所定义的方法，它的值都可以用于这个接口。如果`americanise()`函数执行成功，则返回`nil`值，否则返回相应的`error`值。

```
func filenamesFromCommandLine() (inFilename, outFilename string,
    err error){
    if len(os.Args) > 1 && (os.Args[1] == "-h" || os.Args[1] == "--
help ") {
        err = fmt.Errorf(" usage: %s [<]infile.txt [>]outfile.txt ",
            filepath.Base(os.Args[0]))
        return "", "", err
    }
    if len(os.Args) > 1 {
        inFilename = os.Args[1]
        if len(os.Args) > 2 {
            outFilename = os.Args[2]
        }
    }
```

```

    }
    if inFilename != " " && inFilename == outFilename {
        log.Fatal( " won't overwrite the infile " )
    }
    return inFilename, outFilename, nil
}

```

`filenamesFromCommandLine()`这个函数返回两个字符串和一个错误值。与我们所看到的其他函数不同的是，这里的返回值除了类型外还指定了名字。返回值在函数被执行时先被设置成空值（字符串被设置成空字符串，错误值`err`被设置成`nil`），直到函数体内有赋值语句为其赋值时返回值才改变。（下面讨论`americanise()`函数的时候，我们会更加深入这个主题。）

函数先判断用户是否需要打印帮助信息 [\[13\]](#)。如果是，就用`fmt.Errorf()`函数来创建一个新的`error`值，打印合适的用法，并立即返回。与普通的Go语言代码一样，这个函数也要求调用者检查返回的`error`值，从而做出相应的处理。这也是`main()`函数的做法。`fmt.Errorf()`函数与我们之前所看的`fmt.Printf()`函数类似，不同之处是它返回一个错误值，其中包含由给定的字符串格式和参数生成的字符串，而非将字符串输出到`os.Stdout`中（`errors.New`函数使用一个给定的字符串来生成一个错误值）。

如果用户不需要打印帮助信息，我们再检查他是否输入了命令行参数。如果用户输入了参数，我们将其输入的第一个命令行参数存放到`inFilename`中，将第二个命令行参数存放到`outFilename`中。当然，用户也可能没有输入命令行参数，这样`inFilename`和`outFilename`变量都为空。或者他们也可能只传入了一个参数，其中`inFilename`有文件名而`outFilename`为空。

最后，我们再做一些完整性检查，以保证不会用输出文件来覆盖输入文件，并在必要时退出。如果一切都如预期所料，则正常返回。
[14] 带返回值的函数或方法中必须至少有一个`return`语句。正如在这个函数中所做的一样，给返回值命名，是为了程序清晰，同时也可以用来生成`godoc` 文档。在包含变量名和类型作为返回值的函数或者方法中，使用一个不带返回值的`return` 语句来返回是合法的。在这种情况下，所有返回值变量的值都会被正常返回。本书中我们并不推荐使用不带返回值的`return`语句，因为这是一种不好的Go语言编程风格。

Go语言使用一种非常一致的方式来读写数据。这让我们可以用统一的方式从文件、内存缓冲（即字节或者字符串类型的切片）、标准输入输出或错误流读写数据，甚至也可以用统一的方式从我们的自定义类型读写数据，只要我们自定义的类型实现了相应的读写接口。

一个可读的值必须满足 `io.Reader` 接口。该接口只声明了一个方法 `Read([]byte) (int, error)`。 `Read()`方法从调用该方法的值中读取数据，并将其放到一个字节类型的切片中。它返回成功读到的字节数和一个错误值。如果没有错误发生，则该错误值为`nil`。如果没有错误发生但是已读到文件末尾，则返回 `io.EOF`。如果错误发生，则返回一个非空的错误值。类似的，一个可写的值必须满足 `io.Writer` 接口。该接口也只声明了一个方法 `Write([]byte) (int, error)`。该`Write()`方法将字节类型的切片中的数据写入到调用该方法的值中，然后返回其写入的字节数和一个错误值（如果没有错误发生则其值为`nil`）。

`io`包提供了读写模块，但它们都是非缓冲的，并且只在原始的字节层面上操作。`bufio`包提供了带缓冲的输入输出处理模块，其中的输入模块可作用于任何满足`io.Reader`接口的值（即实现了相应的`Read()`方法），而输出模块则可作用于任何满足 `io.Writer`接口的值（即实现了相应的`Write()`方法）。`bufio` 包的读写模块提供了针对字节或者字符串类型的缓冲机制，因此很适合用于读写UTF-8编码的文本文件。

```

var britishAmerican = " british-american.txt "
func americanise(inFile io.Reader, outFile io.Writer)(err error) {
    reader := bufio.NewReader(inFile)
    writer := bufio.NewWriter(outFile)
    defer func() {
        if err == nil {
            err = writer.Flush()
        }
    }()
    var replacer func(string) string ①
    if replacer, err = makeReplacerFunc(britishAmerican); err != nil {
        return err
    }
    wordRx := regexp.MustCompile( " [A-Za-z]+ " )
    eof := false
    for !eof {
        var line string ②
        line, err = reader.ReadString('\n')
        if err == io.EOF {
            err = nil // 并不是一个真正的
            eof = true // 在下一次迭代这会结束该循环
        } else if err != nil {
            return err // 对于真正的error, 会立即结束
        }
        line = wordRx.ReplaceAllStringFunc(line, replacer)
        if _, err = writer.WriteString(line); err != nil { ③
            return err
        }
    }
}

```

```

    }
}
return nil
}

```

`americanise()`函数为`inFile`和`outFile`分别创建了一个`reader`和`writer`，然后从输入文件中逐行读取数据，然后将所有英式英语词汇替换成等价的美式英语词汇，并将处理结果逐行写入到输出文件中。

只需要往`bufio.NewReader()`函数里传入任何一个实现了`io.Reader`接口的值（即实现了`Read()`方法），就能得到一个带有缓冲的`reader`，`bufio.NewWriter()`函数也类似。需要注意的是，`americanise()`函数不知道也不用关心它从何处读，写向何处，比如 `reader`和`writer`可以是压缩文件、网络连接、字节切片，只要是任何实现`io.Reader`和`io.Writer`接口的值即可。这种处理接口的方式非常灵活，并且使得在Go语言编程中非常易于组合功能。

接下来我们创建一个匿名的延迟函数，它会在`americanise()`函数返回并将控制权交给其调用者之前刷新`writer`的缓冲。这个匿名函数只会在`americanise()`函数正常返回或者异常退出时才执行，由于刷新缓冲区操作也可能会失败，所以我们将 `writer.Flush()`函数的返回值赋值给`err`。如果想忽略任何在刷新操作之前或者在刷新操作过程中发生的任何错误，可以简单地调用`defer writer.Flush()`，但是这样做的话程序对错误的防御性将较低。

Go 语言支持具名返回值，就像我们在之前的`filenamesFromCommandLine()`函数中所做的，在这里我们也充分利用了这个特性（`err error`）。此外，还有一点需要注意的是，在使用具名返回值时有一个作用域的细节。例如，如果已经存在一个名为`value`的返回值，我们可以在函数内的任一位置对该返回值进行赋值，但是如果我们在函数内部某个地方使用了`if value :=...`这样的语句，因为`if`语句

会创建一个新的块，所以这个value是一个新的变量，它会隐藏掉名字同为value的返回值。在americanise()函数中，err是一个具名返回值，因此我们必须保证不使用快速变量声明符:=来为其赋值，以避免意外创建出一个影子变量。基于这样的考虑，我们有时必须在赋值时先声明一个变量，如这里的replacer变量（标识①）和我们这里读入的line变量（标识②）。另一种可选的方式是显式地返回所有返回值，就像我们在其他地方所做的那样。

另外一点需要注意的是，我们在这里使用了空标记符_（标识③）。这里的空标记符作为一个占位符放在需要一个变量的地方，并丢弃掉所有赋给它的值。空占位符不是一个新的变量，因此如果我们使用:=，至少需要声明一个其他的新变量。

Go的标准库中包含一个强大的名为regexp的正则表达式包（参见3.6.5节）。这个包可以用来创建一个指向regexp.Regexp值的指针（即regexp.Regexp类型）。这些值提供了许多供查找和替换的方法。这里我们使用 regexp.Regexp.ReplaceAllStringFunc()方法。它接受一个字符串变量和一个签名为func(string) string的replacer函数作为输入，每发现一个匹配的值就调用一次 replacer 函数，并将该匹配到的文本内容替换为replacer函数返回的文本内容。

如果我们有一个非常小的replacer 函数，比如只是简单地将匹配的字母转换成大写，我们可以在调用替换函数的时候将其创建为一个匿名函数。例如：

```
line = wordRx.ReplaceAllStringFunc(line,  
    func(word string) string {return strings.ToUpper(word)})
```

然而，americanise 程序的replacer 函数虽然也就是几行代码，但它也需要一些准备工作，因此我们创建了一个独立函数makeReplacerFunction()。该函数接受一个包含原始待替换文本的文件

名以及用来替换的文字内容，返回一个`replacer`函数用来执行适当的替换工作。

如果`makeReplacerFunction()`函数返回一个非`nil`的错误值，函数将直接返回。这种情况下调用者需检查所返回的`error`内容并做出相应的处理（如上文所做的那样）。

正则表达式可以使用 `regexp.Compile()`函数来编译。该函数执行成功将返回一个`*regexp.Regexp`值和`nil`，否则返回一个`nil`值和相应的`error`值。这个函数比较适合于正则表达式内容是从外部文件读取或由用户输入的场景，因为需要做一些错误处理。但是这里我们用的是`regexp.MustCompile()`函数，它仅仅返回一个 `*regexp.Regexp`值，或者在正则表达式非法的情况下执行异常流程。示例中所使用的正则表达式尽可能长地匹配一个或者多个英文字母字符。

有了`replacer`函数和正则表达式后，我们开始创建一个无限循环语句，每次循环先从`reader`中读取一行内容。`bufio.Reader.ReadString()`方法将底层`reader`读取过来的原始字节码按UTF-8编码文本的方式读取（严格地讲应该是解码成UTF-8，对于7位的ASCII编码也有效），它最多只能读取指定长度的字节（也可能已读到文件末尾）。该函数将读取的文本内容以方便使用的`string`类型返回，同时返回一个`error`值（不出错误的话为`nil`）。

如果调用 `bufio.Reader.ReadString()`返回的`err` 值非空，可能是读到文件末尾或是读取数据过程中遇到了问题。如果是前者，那么`err`的值应该是`io.EOF`，这是正常的，我们不应该将它作为一个真正的错误来处理，所以这种情况下我们将`err`重新设置为`nil`，并将`eof`设置为`true`以退出循环体。遇到`io.EOF`错误的时候，我们并不立即返回，因为文件的最后一行可能并不是以换行符结尾，在这种情况下我们还需要处理这最后一行文本。

每读到一行，就调用 `regexp.Regexp.ReplaceAllStringFunc()` 方法来处理，并传入这行读取到的文本和对应的 `replacer` 函数。然后我们调用 `bufio.Writer.WriteString()` 方法将处理的结果文本行（可能已经被修改）写入到 `writer` 中。这个 `bufio.Writer.WriteString()` 函数接受一个 `string` 类型的输入，并以 UTF-8 编码的字节流写出到相应目的地，返回成功写出的字节数和一个 `error` 类型值（如果没有发生问题，这个 `error` 类型值将为 `nil`）。这里我们并不关心写入了多少字节，所以用 `_` 把第一返回值忽略掉。如果 `err` 为非空，那么函数将立即返回，调用者会马上接收到相应的错误信息。

正如我们程序中的用法，用 `bufio` 来创建 `reader` 和 `writer` 可以很容易地应用一些字符串处理的高级技巧，完全不用关心原始数据在磁盘上是怎么组织存储的。当然，别忘了我们前面延迟了一个匿名函数，如果没有错误发生所有被缓冲的字节数据都会在 `americanise()` 函数返回时被写入到 `writer` 里。

```
func makeReplacerFunction(file string) (func(string) string, error) {
    rawBytes, err := ioutil.ReadFile(file)
    if err != nil {
        return nil, err
    }
    text := string(rawBytes)
    usForBritish := make(map[string]string)
    lines := strings.Split(text, "\n")
    for _, line := range lines {
        fields := strings.Fields(line)
        if len(fields) == 2 {
            usForBritish[fields[0]] = fields[1]
        }
    }
}
```



```

    }
    return func(word string) string{
        if usWord, found := usForBritish[word]; found {
            return usWord
        }
        return word
    }, nil
}

```

`makeReplacerFunction()`函数接受包含原始字符串和替换字符串文件的文件名作为输入，并返回一个替换函数和一个错误值，这个被返回的替换函数接受一个原始字符串，返回一个被替换的字符串。该函数假设输入的文件是以UTF-8编码的文本文件，其中的每一行使用空格将原始和要替换的单词分隔开来。

除了**bufio**包的**reader**和**writer**之外，Go的**io/ioutil**包也提供了一些使用方便的高级函数，比如我们这里用的**ioutil.ReadFile()**。这个函数将一个文件的内容以**[]byte**值的方式返回，同时返回一个**error**类型的错误值。如果读取出错，返回**nil**和相应的错误，否则，就将它转换成字符串。将UTF-8编码的字节转换成一个字符串是一个非常廉价的操作，因为Go语言中字符串类型的内部表示统一是UTF-8编码的（Go语言的字符串转换内容将在第3章详细阐述）。

由于我们创建的**replacer**函数参数和返回值都是一个字符串，所以我们需要的是—种合适的查找表。Go语言的内置集合类型**map**就非常适合这种情况（参见4.3节）。用**map**来保存键值对，查找速度是很快的，比如我们这里将英式单词作为键，美式单词作为相应的值。

Go语言中的映射、切片和通道都必须通过**make()**函数来创建，并返回一个指向特定类型的值的引用。该引用可以用于传递（如传入到其他函数），并且在被引用的值上做的任何改变对于任何访问该值的

代码而言都是可见的。在这里我们创建了一个名为 `usForBritish` 的空映射，它的键和值都是字符串类型。

在映射创建完成后，我们调用 `strings.Split()` 函数将文件的内容（就是一个字符串）使用分隔符“`\n`”切分为若干个文本行。这个函数的输入参数为一个字符串和一个分隔符，会对输入的字符串进行尽可能多次数的切分（如果我们想限制切分的次数，可以使用 `strings.SplitN()` 函数）。

我们使用一个之前没有接触过的 `for` 循环语法来遍历每一行，这一次我们使用的是一个 `range` 语句。这种语法用来遍历映射中的键值对非常方便，可用于读取通道的元素，另外也可用于遍历切片或者数组。当我们使用切片（或数组）时，每次迭代返回的是切片的索引和在该索引上的元素值，其索引从0开始（如果该切片为非空的话）。在本例中，我们使用循环来迭代每一行，但由于我们并不关心每一行的索引，所以用了一个 `_` 占位符把它忽略掉。

我们需要将每行切分成两部分：原始字符串和替换的字符串。我们可以使用 `strings.Split()` 函数，但它要求声明一个确定的分隔符，如“`"`”，这在某些手动分隔的文件中可能失败，因为用户可能意外地输入多个空格或者使用制表符来代替空格。幸亏 Go 语言标准库提供了另一个 `strings.Fields()` 函数以空白分隔符来分隔字符串，因此能更恰当地处理用户手动编辑的文本。

如果变量 `fields`（其类型为 `[]string`）恰好有两个元素，我们将对应的“键值”对插入映射中。一旦该映射的内容准备好，我们就可以开始创建用来返回给调用者的 `replacer` 函数。

我们将 `replacer` 函数创建为匿名函数，并将其当做一个参数来让 `return` 语句返回，该 `return` 语句同时返回一个空的错误值（当然，我们本来可以更繁琐点，将该匿名函数赋值给一个变量，并将该变量返

回)。这个匿名函数的签名与`regexp.Regexp.ReplaceAllStringFun()`方法所期望传入的函数签名必须完全一致。

我们在匿名函数`replacer`里所做的只是查找一个给定的单词。如果我们在左边通过一个变量来获取一个映射的元素，该元素将被赋值给对应的变量。如果映射中对应的键不存在，那么所获取的值为该类型的空值。如果该映射值类型的空值本身也是一个合法的值，那我们还能如何判断一个给定的值是否在映射中呢？Go语言为此提供了一种语法，即赋值语句的左边同时为两个变量赋值，第一个变量用来接收该值，第二个变量用来接收一个布尔值，表示该键在映射中是否找到。如果我们只是想知道某个特定的值是否在映射中，该方法通常有效。本例中我们在 `if` 语句中使用第二种形式，其中有一个简单的语句（一个简短的变量声明）和一个条件（那个布尔变量`found`）。因此，我们得到`usWord`变量（如果所给出的单词不在映射中，该变量的值为空字符串）和一个布尔类型的`found`标志。如果英式英语的单词找到了，我们返回相应的美式英语单词；否则，我们简单地将原始单词原封不动地返回。

我们从`makeReplacerFunction()`函数中还可以发现一个有些微妙的地方。在匿名函数内部我们访问了在匿名函数的外层创建的`usForBritish` 变量（是一个映射）。之所以可以这么做，是因为Go支持闭包（参见5.6.3节）。闭包是一个能够“捕获”一些外部状态的函数，例如可以捕获创建该函数的函数的某些状态，或者闭包所捕获的该状态的任意一部分。因此在这里，在函数`makeReplacerFunction()`内部创建的匿名函数是一个闭包，它捕获了`usForBritish`变量。

还有一个微妙的地方就是，`usForBritish`本应该是一个本地变量，然而我们却可以在它被声明的函数之外使用它。在 Go语言中完全可以返回本地变量。即使是引用或者指针，如果还在被使用，Go语言并不会删除它们，只有在它们不再被使用时（也就是当任何保存、引用或

者指向它们的变量超出作用域范围时）才用垃圾回收机制将它们回收。

本节给出了一些利用`os.Open()`、`os.Create()`和`ioutil.ReadFile()`函数来处理文件的基础和高级功能。在第8章中我们将介绍更多的文件处理相关内容，包括读写文本文件、二进制文件、JSON文件和XML文件。Go语言的内置集合类型如切片和映射提供了非常良好的性能和极大的便利性，帮助开发者大大降低了创建自定义类型的需求。我们将在第4章详细阐述Go语言的集合类型。Go语言将函数当做一类值来对待并支持闭包，使得开发者在写程序时可以使用一些高级而非常有用的编程技巧。同时，Go语言的`defer`语句能非常直接简单明了地避免资源泄露。

1.7 从极坐标到笛卡儿坐标——并发

Go语言的一个关键特性在于其充分利用现代计算机的多处理器和多核的功能，且无需给程序员带来太大负担。完全无需任何显式锁就可写出许多并发程序（虽然Go语言也提供了锁原语以便在底层代码需要用到时使用，我们将在第7章中详细阐述）。

Go语言有两个特性使得用它来做并发编程非常轻松。第一，无需继承什么“线程”（`thread`）类（这在Go语言中其实也不可能）即可轻易地创建`goroutine`（实际上是非常轻量级的线程或者协程）。第二，通道（`channel`）为`goroutine`之间提供了类型安全的单向或者双向通信，这也可以用来同步`goroutine`。

Go语言处理并发的方式是传递数据，而非共享数据。这使得与使用传统的线程和锁方式相比，用Go语言来编写并发程序更为简单。由

于没有使用共享数据，我们不会进入竞态条件（例如死锁），我们也不必记住何时该加锁和解锁，因为没有共享的数据需要保护。

本节中，我们会看看本章中的第五个也是最后一个“概览”示例。这节的例子使用两个通信通道，并且在一个独立的goroutine中处理数据。对于这样一种小巧的程序而言，这显然是大材小用，但这样做的目的是为了以尽量简洁的方式来讲解这些Go语言功能的基本使用方式。我们将在第7章展示一个更加实用的并发示例，它会给出许多一起使用通道和goroutine的不同做法。

我们将要讲解的这个程序叫做polar2cartesian。这是一个交互型的命令行程序，首先提示用户输入两个由空格分隔的数字：一个半径和一个角度，然后该程序使用它们来计算相应的笛卡儿坐标。除了会介绍一种并发编程的特定实现方式，这个示例也展示一些简单的结构体（struct）类型，以及如何确定程序是运行在一个类Unix系统上还是运行在Windows系统上，因为两个系统的不同点值得关注。这里有一个在Linux的终端下运行的示例程序：

```
./polar2cartesian
```

```
Enter a radius and an angle (in degrees), e.g., 12.5 90, or Ctrl+D to quit.
```

```
Radius and angle: 5 30.5
```

```
Polar radius=5.00  $\theta=30.50^\circ$  → Cartesian x=4.31 y=2.54
```

```
Radius and angle: 5 -30.25
```

```
Polar radius=5.00  $\theta=-30.25^\circ$  → Cartesian x=4.32 y=-2.52
```

```
Radius and angle: 1.0 90
```

```
Polar radius=1.00  $\theta=90.00^\circ$  → Cartesian x=-0.00 y=1.00
```

```
Radius and angle: ^D
```

```
$
```

这个程序的源文件位于 `polar2cartesian/polar2cartesian.go`，我们将自上而下地解读它，先是导入包，我们用到结构体（`struct`），接着是 `init()` 函数、`main()` 函数，然后是被 `main()` 函数调用的函数等。

```
import (  
    "bufio "  
    "fmt "  
    "math "  
    "os "  
    "runtime "  
)
```

这是 `polar2cartesian` 程序导入的几个包，其中有些在前面几节中提到过，因此我们只在这里提提新引入的包。`math` 包提供了操作浮点数的数学函数（参见2.3.2节），而 `runtime` 包提供了一些运行时控制，例如可以知道该程序运行在哪个平台上。

```
type polar struct {  
    radius  float64  
    theta   float64  
}  
  
type cartesian struct {  
    x  float64  
    y  float64  
}
```

Go语言的结构体是一种能够用来保存（聚合或者嵌入）一个或者多个数据字段的类型。这些字段可以是像本例所采用的内置类型（`float64`）、结构体、接口，或者所有这些类型的组合。（一个接口类型的数据字段其实只是一个指向任意类型值的指针，该类型实现了这个接口，也就是实现了该接口所声明的所有方法。）

我们很自然地使用了小写的希腊字母 θ 来表示极坐标的角度，这在 Go 语言中很容易做到，因为 Go 语言支持 UTF-8 编码的字符。Go 语言允许我们使用任何 Unicode 字符作为我们的标识符，而限于英文字母。

虽然这两个结构体恰好包含了完全相同的字段类型，但它们仍属不同类型，两者之间也不能自动地相互转换。这也可以认为是防御性编程，毕竟用一个极坐标来代替一个笛卡儿坐标也不合理。在有些情况下这种转换是有意义的，这样我们可以轻易地创建一个转换方法（也就是该类型的某个方法可以返回另一个类型），它能够充分利用 Go 语言的组合特性来从一个源类型创建另一个目标类型（数值数据类型的转换将在第2章中详述。字符串类型的转换将在第3章中详述）。

```
var prompt = " Enter a radius and an angle (in degrees), e.g., 12.5 90, "
            + " or %s to quit. "

func init() {
    if runtime.GOOS == " windows " {
        prompt = fmt.Sprintf(prompt, " Ctrl+Z, Enter ")
    } else { //类Unix
        prompt = fmt.Sprintf(prompt, " Ctrl+D ")
    }
}
```

如果一个包里包含了一个或多个 `init()` 函数，那么它们会在 `main()` 函数之前被自动执行，而且 `init()` 函数不能被显式调用。因此当我们的 `polar2cartesian` 程序启动时，这个 `init()` 函数会首先被调用。这里我们使用不同的 `init()` 函数来为不同的平台设置不同的提示信息，因为不同的平台文件结束的标志是不同的，例如在 Windows 平台上是使用 `Ctrl+Z` 然后按回车键来结束文件。 `runtime` 包提供了一个字符串类型的常量

GOOS来标示程序所运行的操作系统，其常用值为darwin（Mac OS X）、freebsd、linux以及windows。

在深入剖析 `main()` 函数以及剩下的程序之前，让我们先简单地介绍一下通道，并在使用它之前看一些好玩的小示例。

通道是基于 Unix 上管道的思想而被设计出来的，它提供了双向（或者如我们这里用到的单向）数据通信。通道的行为跟FIFO（先进先出）队列一样，因此它们会保留发送给它们的数据的先后顺序。通道中的数据不能被删除，但我们可以随便忽略任何或者所有接收到的数据。让我们看一个非常简单的例子，首先我们创建一个通道：

```
messages := make(chan string, 10)
```

我们使用 `make()` 函数来创建一个通道，其声明的语法为 `chan Type`。这里我们创建了一个名为 `messages` 的通道，用来发送和接收字符串消息。`make()` 函数的第二个参数是通道缓冲区的大小（其默认值为 0）。这里我们将其设置得足够大，以便能够容纳 10 个字符串。如果通道的缓冲区满了，就会发生阻塞，直到其中的至少一个项被接收。这也意味着可以向一个通道传入任意数量的项，因为其中的数据会不断地被取回，而给后面的数据腾出足够的空间。如果另一端在等待接收一个数据，那么一个缓冲大小为0的缓冲只可以发送一个数据（也可以使用Go语言的`select`语句来得到非阻塞通道的效果，我们将在第7章阐述）。

现在，让我们来发送一些字符串到通道里：

```
messages <- " Leader "
```

```
messages <- " Follower "
```

当`<-`通信操作符用做二元操作符时，它的左操作数必须是一个通道，右操作数必须是发往该通道的数据，其类型为通道声明时所能接收的类型。这里，我们先将字符串`Leader`发往通道，然后再将字符串`Follower`发往通道。


```
message1 := <-messages
```

```
message2 := <-messages
```

当<-通信操作符用做一元操作符时只有一个左操作数（必须是一个通道），它是一个接收器，一直阻塞直到获得一个可以返回的数据。这里，我们从该messages通道中取回两条消息。字符串Leader被赋值给变量message1，字符串Follower被赋值给变量message2，这两个变量都是字符串类型的。

通常情况下通道用于goroutine之间的通信。通道在发送和接收数据时无需加锁，而其阻塞的特性可以用于达到线程同步的效果。

我们已经了解了一些关于通道的基本知识，现在让我们在实际代码中看看通道和goroutine的使用。

```
func main() {  
    questions := make(chan polar)  
    defer close(questions)  
    answers := createSolver(questions)  
    defer close(answers)  
    interact(questions, answers)  
}
```

一旦有任一init()函数返回，Go语言的运行时系统就会调用main包的main()函数。

在这个示例里，main()函数先创建了一个用来传输polar结构体信息的通道（通道类型为chan polar），然后将其赋给questions变量。一旦通道创建好之后，我们使用defer语句调用内置的close()函数来保证在该通道被使用完毕之后能被正常关闭。接下来我们调用createSolver()函数，将questions传递给它，返回一个名为answers的通道用于接收消息。我们使用另一个defer语句来保证answers在使用完后

能够被正常关闭。最后我们将这两个通道传递给`interact()`函数，接下来的工作就交给用户交互了。

```
func createSolver(questions chan polar) chan cartesian {
    answers := make(chan cartesian)
    go func() {
        for {
            polarCoord := <-questions①
             $\theta := \text{polarCoord}.\theta * \text{math.Pi} / 180.0$  // 度变弧度
            x := polarCoord.radius * math.Cos( $\theta$ )
            y := polarCoord.radius * math.Sin( $\theta$ )
            answers <- cartesian{x, y} ②
        }
    }()
    return answers
}
```

`createSolver()`函数首先创建了一个名为`answers`的通道，然后往里面发送接收到的问题（极坐标）的答案（笛卡儿坐标）。

在通道创建后，该函数然后调用了一个`go`语句。`go`语句接受一个函数调用（这种语法类似于`defer`语句），这会创建一个独立的异步`goroutine`来执行这个函数。这也意味着当前函数的控制流程会继续向下执行，比如我们这里`go`语句之后就是一个`return`语句，它将`answers`返回给调用者。前面我们已经知道，在Go语言里返回本地变量是非常安全的，因为Go语言会为我们打理一切内存管理的杂事。

在这个 `go` 语句里我们创建了一个匿名函数，该函数有一个无限循环体处于阻塞等待状态（但不会阻塞其他`goroutine`，也不会阻塞创建该`goroutine`的函数），直到它接收到一个问题（本例中是一个定义在`polar`结构体上的`questions`通道）。当收到一个极坐标时，该匿名函数

通过一定的数学计算（使用标准库中的`math`包）得出相应的笛卡儿坐标，然后使用Go语言的组合语法将其结果创建成为一个`cartesian`结构体发送给`answers`。

在语句①中，`<-`操作符作为一元操作符使用，它从`questions`通道中获取一个极坐标。而语句②则作为二元运算符使用，它的左操作数是用于接收数据的`answers`通道，右操作数则是用于发送数据的`cartesian`结构体。

一旦对函数 `createSolver()` 的调用完成，我们就有了两个通信通道，还有一个独立的`goroutine`用于等待极坐标发送到`questions`上，而其他包括执行`main()`函数在内的`goroutine`则不会阻塞。

```
const result = " Polar radius=%.02f  $\theta$ =%.02f $^\circ$   $\rightarrow$  Cartesian x=%.02f y=%.02f\n "
```

```
func interact(questions chan polar, answers chan cartesian) {
    reader := bufio.NewReader(os.Stdin)
    fmt.Println(prompt)
    for {
        fmt.Println( " Radius and angle: " )
        line, err := reader.ReadString('\n')
        if err != nil {
            break
        }
        var radius,  $\theta$ float64
        if _, err := fmt.Sscan(line, " %f %f ", &radius, & $\theta$ ); err != nil {
            fmt.Println(os.Stderr, " invalid input " )
            continue
        }
        questions <- polar{radius,  $\theta$ }
```

```

        coord := <-answers
        fmt.Printf(result, radius,  $\theta$ , coord.x, coord.y)
    }
    fmt.Println()
}

```

调用这个函数时需传入两个通道作为参数。由于我们需要在控制台上跟用户交互，因此该函数开始处为`os.Stdin`创建了一个带缓冲的`reader`。然后打印提示符告诉用户输入什么，怎样输入，以及怎样退出。如果用户只按了一个回车键（没有输入任何数字），那么我们就直接退出程序，而不是还让用户输入文件的结束符。然而，通过要求用户输入文件结束符，我们可以使得`polar2cartesian`程序更加灵活，因为这样就可以从任意的外部文件中获得输入了（假设输入的文件每行只有两个由空格分隔的数字）。

随后函数就进入了无限循环，提示用户输入极坐标（一个半径和一个角）。要求用户输入数据后，函数会等待用户输入某些文字然后再按回车键，或者按 `Ctrl+D` 键（在 `Windows` 上是按`Ctrl+Z`和回车键）来表示用户输入结束。我们并没有检查返回的错误值，如果它不为`nil`，我们就退出循环并返回到调用者 `main()`函数，然后 `main()`函数会随后退出（同时调用它的延迟执行语句来关闭通道）。

我们创建了两个`float64`类型的变量来保存用户输入，然后使用`fmt.Sscanf()`函数来解析每一行。该函数接受一个字符串作为待解析的输入字符串、字符串的解析格式（在本例中是两个由空格分隔的浮点数），后面紧跟的是一个或者多个用来填充的参数（地址操作符 `&` 用于得到指向一个变量的指针。参见 4.1 节）。该函数返回其成功解析的元素数量和一个 `error`值（或者为`nil`）。万一发生错误，我们将错误信息打印到`os.Stderr`，这样可以使得即使将程序的`os.Stdout`重定向到一

个文件，其错误信息在控制台也能够看到。Go语言的这些强大而又灵活的扫描函数将在第8章中详细阐述（参见8.1.3.2节）以及表8-2。

如果用户输入了合法的数字并已经以polar结构体的格式发送到questions通道，那么就会阻塞主 goroutine，等待 answers 通道的响应。createSolver()函数额外创建的一个goroutine会阻塞等待questions通道接收到一个polar类型的数据，因此当我们发送polar数据后，这个goroutine将执行计算，并将计算结果cartesian发送回answers通道，然后等待另一个问题的输入（阻塞其自身）。一旦 interact()函数在 answers 通道上接收到cartesian，interact()就不再阻塞。这样，我们就使用 fmt.Printf()函数打印结果信息，并将极坐标和笛卡儿坐标的值输入作为结果字符串的%占位符。这些goroutine和这些通道的关系可以通过图1-1来解释。

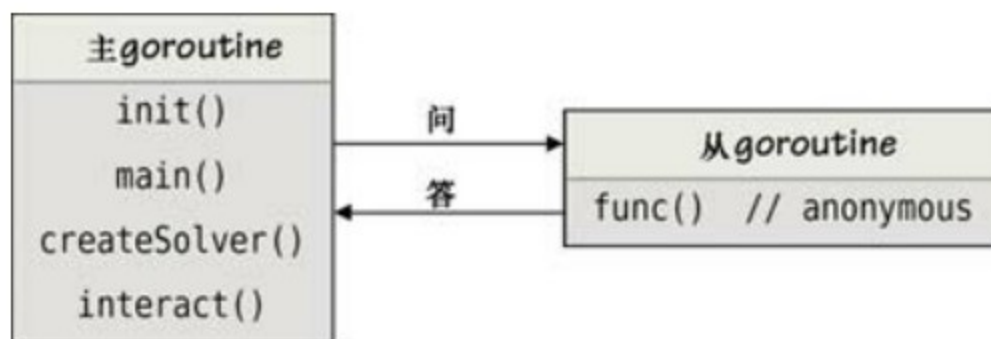


图1-1 两个相互通信的goroutine

interact()函数中的for 循环是一个无限循环，在打印一个结果后又让用户输入下一个半径和角度值，直到用户输入了一个文件结束符。这个输入可能来自于用户的交互输入，也可能是因为到达了一个重定向输入文件的末尾。

程序 polar2cartesian 中的计算非常轻量，因此没必要在另一个独立的goroutine中执行。然而如果一个程序需要做多个相互独立的大规模计算以作为一个输入的结果，使用本文所用到的方法可能会更好，

比如为每一个计算都创建一个独立的goroutine。我们会在第7章看到关于通道和goroutine的更加实用的示例。

通过讲解本章中所给出的5个小示例程序，我们完成了对Go语言特性的概述。当然，我们将在本书的后面章节看到，Go语言所提供的远远不止这一章所能写下的。接下来的每一章将专注于Go语言的某一特定主题，以及与该主题相关的标准库。这章的结尾处有个小习题，其量虽小，但是需要些思考和细心才能完成。

1.8 练习

将 bigdigits 文件夹复制为另一个文件夹，比如命名为 my_bigdigits，修改 my_bigdigits/big-digits.go 的内容以使得新版本的 bigdigits 程序可以可选地输出由一条“*”组成的上横线和下横线，并且还带有改进过的命令行参数处理能力。

如果运行程序时没有输入数字作为命令行参数，原来版本的程序会输出其使用信息。请更改该程序，使得用户使用 -h 或者 --help 参数的时候程序也能输出使用信息。例如：

```
./bigdigits --help
usage: bigdigits [-b|--bar] <whole-number>
-b --bar draw an underbar and an overbar
```

如果运行时没有提供 --bar（或者 -b）选项，那么程序的功能应该与原来的版本一样。下面是在给出该参数的情况下程序的预期输出：

```
./bigdigits --bar 8467243
*****
888      4      666   77777   222      4      333
```

```

8 8 44 6 7 2 2 44 3 3
8 8 4 4 6 7 2 4 4 3
888 4 4 6666 7 2 4 4 33
8 8 444444 6 6 7 2 444444 3
8 8 4 6 6 7 2 4 3 3
888 4 666 7 22222 4 333
*****

```

虽然与之前的版本相比，只需更改几行代码就可以在第一行之前输出上横线以及在最后一行后面输出下横线，但该方案需要更加精细的，命令行处理。总而言之，该方案大概需要 20 行的额外代码，其 `main()` 函数的长度需要增加到大概原先的两倍（大约 40 行），其中大部分代码与处理命令行有关。文件 `bigdigits_ans/bigdigits.go` 提供了一个参考答案。

提示：为了防止输出的横线过长，该解决方案与之前输出每行数字的方式稍微有点不同。同时，该解答中需要导入 `strings` 包并使用其中的 `strings.Repeat(string, int)` 函数。该函数返回一个字符串，该字符串是由该函数的第一个参数重复第二个参数所指定的次数生成的。何不在本地（参见 1.1 节中的“Go 语言官方文档”）或者到 golang.org/pkg/strings 查找一下这个函数，开始熟悉下 Go 语言标准库文档呢？

使用专为处理命令行参数设置的包可以带来很大的便利性。Go 语言标准库中既包含一个相对基本的命令行解析包 `flag` 以支持 X11 风格的选项（也就是 `-option` 这样的选项）。另外，godashboard.appspot.com/project 上也有许多可选的支持 GNU 风格的短选项或长选项（也就是 `-o` 和 `--option` 这样的形式）的命令行参数解析器。

[1].有些Windows编辑器（如记事本）不符合Unicode编码标准，会在UTF-8编码的文件开始处插入0xEF、0xBB和0xBF字节。本书的示例假设所有的UTF-8文件中不包含这些字节。

[2].由于本书假设使用gc编译器，使用gccgo的读者需要按照链接golang.org/doc/gccgo_install.html中描述的编译和链接过程来进行。类似地，使用其他编译器的读者也需要按照相应的编译器指南来进行编译和链接。

[3].从现在开始我们只给出类Unix系统的命令行，并且假设Windows程序员能够自行转换过来。

[4].为了在文中更好的引用，我们在程序中使用一些简单的语法高亮，有时会高亮一些代码行或者以数字（①，②，...）来对代码行进行标注。这些都不是Go语言的组成部分。

[5].与C、C++以及Java语言不同的是，Go语言中的++和--操作符只可用于语句而非表达式中。更进一步讲，它们可能只可用做后缀操作符而非前缀操作符。这意味着，某些特定顺序的求值问题不会在Go语言中出现——因为，谢天谢地，Go语言中不会写出f(i++)以及A[i]=b[++i]这样的表达式来。

[6].不同于C++，Go语言的struct（结构体）不是伪类。例如，Go语言的struct结构体支持组合与委派，但不支持继承。

[7].Go语言的空接口扮演的是Java中的Object或者C/C++中的void*类型一样的角色。

[8].在其他语言中，接收器一般被称为this或self，使用这种称谓在Go语言中也没问题，但被认为是不太好的Go语言风格。

[9].Go语言中的指针除了不支持指针运算之外（其实也不需要），其他的与C和C++里的是是一样的。

[10].Go语言中的nil与C、C++中的NULL或0，或者Java中的null以及Objective-C中的nil是等价的。

[11].这是Go语言不需要类似于C、C++语言中的-> 操作符的原因。

[12].Go语言的错误处理与C++、Java和Python非常不同，因为这几门语言经常将异常处理机制同时用于处理错误和异常。关于Go语言panic/recover机制的讨论和原理阐述请参见https://groups.google.com/group/golang-nuts/browse_thread/thread/1ce5cd050bb973e4?pli=1。

[13].Go的标准库包含有一个flag包用于命令行参数处理。GNU格式命令行参数处理的相关第三方包可以从godashboard.appspot.com/project获取（使用第三方包的内容将在第9章中阐述）。

[14].实际上用户还是可以使用重定向来覆写输入文件的，例如[\\$./americanise infile > infile](#)，但是我们至少防止了很明显的意外。

第2章 布尔与数值类型

这是关于过程式编程的四章内容中的第一章，它构成了 Go语言编程的基础——无论是过程式编程、面向对象编程、并发编程，还是这些编程方式的任何组合。

本章涵盖了Go语言内置的布尔类型和数值类型，同时简要介绍了一下Go标准库中的数值类型。本章将介绍，除了各种数值类型之间需要进行显式类型转换以及内置了复数类型外，从C、C++以及Java等语言转过来的程序员还会有更多惊喜。

本章第一小节讲解了Go语言的基础，比如如何写注释，Go语言的关键字和操作符，一个合法标识符的构成，等等。一旦这些基础性的东西讲解完后，接下来的小节将讲解布尔类型、整型以及浮点型，之后也对复数进行了介绍。

2.1 基础

Go语言支持两种类型的注释，都是从C++借鉴而来的。行注释以//开始，直到出现换行符时结束。行注释被编译器简单当做一个换行符。块注释以/*开头，以*/结尾，可能包含多个行。如果块注释只占用了一行（即/* inline comment*/），编译器把它当做一个空格，但是如果该块注释占用了多行，编译器就把它当做一个换行符。（我们将在第5章看到，换行符在Go语言中非常重要。）

Go标识符是一个非空的字母或数字串，其中第一个字符必须是字母，该标识符也不能是关键字的名字。字母可以是一个下划线_，或者Unicode 编码分类中的任何字符，如大写字母“Lu”（letter，uppercase）、小写字母“Ll”（letter，lowercase）、首字母大写“Lt”（letter，titlecase）、修饰符字母“Lm”（letter, modifier）或者其他字母，“Lo”（letter，other）。这些字符包含所有的英文字母（A~Z以及a~z）。数字则是Unicode编码 " Nd " 分类（number, decimal digit）中的任何字符，这些字符包括阿拉伯数字 0~9。编译器不允许使用与某个关键字（见表 2-1）一样的名字作为标识符。

表2-1 Go语言的关键字

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var

Go语言预先定义了许多标识符（见表 2-2），虽然可以定义与这些预定义的标识符名字一样的标识符，但是这样做通常很不明智。

表2-2 Go语言预定义的标识符

append	copy	int8	nil	true
bool	delete	int16	Panic	uint
byte	error	int32	print	uint8
cap	false	int64	println	uint16
close	float32	iota	real	uint32
complex	float64	len	recover	uint64
complex64	imag	make	rune	uintptr
complex128	int	new	string	

标识符都是区分大小写的，因此 LINECOUNT、Linecount、LineCount、lineCount和linecount是5个不一样的标识符。以大写字母开

头的标识符，即Unicode分类中属于“Lu”的字母（包含A～Z），是公开的——以Go语言的术语来说就是导出的，而任何其他标识符都是私有的——用Go语言的术语来说就是未导出的。（这项规则不适用于包的名字，包名约定为全小写。）第6章讨论面向对象编程以及第9章讨论包时，我们会在实际的代码中看到这两者的区别。

空标识符“_”是一个占位符，它用于在赋值操作的时候将某个值赋值给空标识符，从而达到丢弃该值的目的。空标识符不是一个新的变量，因此将它用于:=操作符的时候，必须同时为至少另一个值赋值。通过将函数的某个甚至是所有返回值赋值给空标识符的形式将其丢弃是合法的。然而，如果不需要得到函数的任何返回值，更为方便的做法是简单地忽略它。这里有些例子：

```
count, err = fmt.Println(x)    // 获取打印的字节数以及相应的error值
```

```
count, _ = fmt.Println(x)      // 获取打印的字节数，丢弃error值
_, err = fmt.Println(x)        // 丢弃所打印的字节数，并返回error值
```

```
fmt.Println(x)                 // 忽略所有返回值
```

打印到终端的时候忽略返回值很常见，但是使用fmt.Fprint()以及类似函数打印到文件和网络连接等情况时，则应该检查返回的错误值。

（Go语言的打印函数将在3.5节详细介绍。）

常量和变量

常量使用关键字const声明；变量可以使用关键字var声明，也可以使用快捷变量声明语法。Go语言可以自动推断出所声明变量的类型，但是如果需要，显式指定其类型也是合法的，比如声明一种与Go语言的常规推断不同的类型。下面是一些声明的例子：

```
const limit = 512              // 常量，其类型兼容任何数字
const top uint16 = 1421        // 常量，类型：uint16
```

```
start := -19           // 变量，推断类型: int
end := int64(9876543210) // 变量，类型: int64
var i int              // 变量，值为0，类型: int
var debug = false      // 变量，推断类型: bool
checkResults := true   // 变量，推断类型: bool
stepSize := 1.5        // 变量，推断类型: float64
acronym := "FOSS"      // 变量，推断类型: string
```

对于整型字面量 Go语言推断其类型为 `int`，对于浮点型字面量 Go语言推断其类型为 `float64`，对于复数字面量 Go语言推断其类型为 `complex128`（名字上的数字代表它们所占的位数）。通常的做法是不去显式地声明其类型，除非我们需要使用一个Go语言无法推断的特殊类型。这点我们会在 2.3 节中讨论。指定类型的数值常量（即这里的 `top`）只可用于别的数值类型相同的表达式中（除非经过转换）。未指定类型的数值常量可用于别的数值类型为任何内置类型的表达式中（例如，常量 `limit` 可以用于包含整型或者浮点型数值的表达式中）。

变量 `i` 并没有显式的初始化。这在Go语言中非常安全，因为如果没有显式初始化，Go语言总是会将零值赋值给该变量。这意味着每一个数值变量的默认值都保证为 `0`，而每个字符串都默认为空。这可以保证Go程序避免遭受其他语言中的未初始化的垃圾值之灾。

枚举

需要设置多个常量的时候，我们不必重复使用 `const` 关键字，只需使用 `const` 关键字一次就可以将所有常量声明组合在一起。（第1章中我们导入包的时候使用了相同的语法。该语法也可以用于使用 `var` 关键字来声明一组变量。）如果我们只希望所声明的常量值不同，并不关心其值是多少，那么可以使用Go语言中相对比较简陋的枚举语法。

<pre>const Cyan = 0 const Magenta = 1 const Yellow = 2</pre>	<pre>const (Cyan = 0 Magenta = 1 Yellow = 2)</pre>	<pre>const (Cyan = iota //0 Magenta //1 Yellow //2)</pre>
--	--	---

这3个代码片段的作用完全一样。声明一组常量的方式是，如果第一个常量的值没有被显式设置（设为一个值或者是*iota*），则它的值为零值，第二个以及随后的常量值则设为前面一个常量的值，或者如果前面常量的值为*iota*，则将其后续值也设为*iota*。后续的每一个*iota*值都比前面的*iota*值大1。

更正式的，使用*iota*预定义的标识符表示连续的无类型整数常量。每次关键字**const**出现时，它的值重设为零值（因此，每次都会定义一组新的常量），而每个常量的声明的增量为1。因此在最右边的代码片段中，所有常量（指**Magenta**和**Yellow**）都被设为*iota*值。由于**Cyan**紧跟着一个**const**关键字，其*iota*值重设为0，即**Cyan**的值。**Magenta**的值也设为*iota*，但是这里*iota*的值为1。类似地，**Yellow**的值也是*iota*，它的值为2。而且，如果我们在其末尾再添加一个**Black**（在**const**组内部），它的值就被隐式地设为*iota*，这时它的值就是3。

另一方面，如果最右边的代码片段中没有*iota*标识符，**Cyan**就会被设为0，而**Magenta**的值则会设为**Cyan**的值，**Yellow**的值则被设为**Magenta**的值，因此最后它们都被设为零值。类似的，如果**Cyan**被设为9，那么随后的值也会被设为9。或者，如果**Magenta**的值设为5，**Cyan**的值就被设为0（因为是组中的第一个值，并且没有被设为一个显式的值或者*iota*），**Magenta**的值就是5（显式地设置），而**Yellow**的值也是5（前一个常量的值）。

也可以将*iota*与浮点数、表达式以及自定义类型一起使用。

```
type BitFlag int
```

```
const (
```

```
    Active BitFlag = 1 << iota
```

```
// 1 << 0 == 1
```

```

    Send    // 隐式地设置成BitFlag = 1 << iota    // 1 << 1 == 2
    Receive // 隐式地设置成BitFlag = 1 << iota    // 1 << 2 == 4
)
flag := Active | Send

```

在这个代码片段中，我们创建了3个自定义类型**BitFlag**的位标识，并将变量**flag**（其类型为**BitFlag**）的值设为其中两个值的按位或（因此**flag**的值为3，Go语言的按位操作符已在表2-6中给出）。我们可以略去自定义类型，这样Go语言就会认为定义的常量是无类型整数，并将**flag**的类型推断成整型。**BitFlag**类型的变量可以保存任何整型值，然而由于**BitFlag**是一个不同的类型，因此只有将其转换成**int**型后才能将其与**int**型数据一起操作（或者将**int**型数据转换成**BitFlag**类型数据）。

正如这里所表示的，**BitFlag**类型非常有用，但是用来调试不太方便。如果我们打印**flag**的值，那么得到的只是一个3，没有任何标记表示这是什么意思。Go语言很容易控制自定义类型的值如何打印，因为如果某个类型定义了**String()**方法，那么**fmt**包中的打印函数就会使用它来进行打印。因此，为了让**BitFlag**类型可以打印出更多的信息，我们可以给该类型添加一个简单的**String()**方法。（自定义类型和方法的内容将在第6章详细阐述。）

```

func (flag BitFlag) String() string {
    var flags []string
    if flag & Active == Active {
        flags = append(flags, "Active")
    }
    if flag & Send == Send {
        flags = append(flags, "Send")
    }
    if flag & Receive == Receive {

```

```

        flags = append(flags, "Receive")
    }
    if len(flags) > 0 { // 在这里，int(flag)用于防止无限循环，至关重要！
        return fmt.Sprintf("%d(%s)", int(flag), strings.Join(flags, "|"))
    }
    return "0()"
}

```

对于已设置好值的位域，该方法构建了一个（可能为空的）字符串切片，并将其以十进制整型表示的位域的值以及表示该值的字符串打印出来。（通过将%d标识符设为%b，我们可以轻易地将该值以二进制整数打印出来。）正如其中的注释所说，当将flag传递给fmt.Sprintf()函数的时候，将其类型转换成底层的int类型至关重要，否则BitFlag.String()方法会在flag上递归地调用，这样就会导致无限的递归调用。（内置的append()函数将在4.2.3节中讲解。fmt.Sprintf()和strings.Join()函数将在第3章讲解。）

```
Println(BitFlag(0), Active, Send, flag, Receive, flag|Receive)
```

0()	1(Active)	2(Send)	3(Active Send)	4(Receive)
7(Active Send Receive)				

上面的代码片段给出了带String()方法的BitFlag类型的打印结果。很明显，与打印纯整数相比，这样的打印结果对于调试代码更有用。

当然，也可以创建表示某个特定范围内的整数的自定义类型，以便创建一个更加精细的自定义枚举类型，我们会在第6章详细阐述自定义类型的内容。Go语言中关于枚举的极简方式是Go哲学的典型：Go语言的目标是为程序员提供他们所需要的一切，包括许多强大而方便的

特性，同时又让该语言尽可能地保持简小、连贯而且快速编译和运行。

2.2 布尔值和布尔表达式

Go语言提供了内置的布尔值true和false。Go语言支持标准的逻辑和比较操作，这些操作的结果都是布尔值，如表2-3所示。

表2-3 布尔值和比较操作符

语法	描述/结果
<code>!b</code>	逻辑非操作符，如果表达式 <code>b</code> 的值为 <code>true</code> ，则操作结果为 <code>false</code>
<code>a b</code>	短路逻辑或操作符，只要布尔表达式 <code>a</code> 或者 <code>b</code> 中的任何一个表达式为 <code>true</code> ，表达式的结果都为 <code>true</code>
<code>a && b</code>	短路逻辑与操作符，如果两个布尔表达式 <code>a</code> 和 <code>b</code> 都为 <code>true</code> ，则整个表达式的值为 <code>true</code>
<code>x < y</code>	如果表达式 <code>x</code> 的值小于表达式 <code>y</code> 的值，则表达式的结果为 <code>true</code>
<code>x <= y</code>	如果表达式 <code>x</code> 的值小于或者等于表达式 <code>y</code> 的值，则表达式的结果为 <code>true</code>

续表

语法	描述/结果
<code>x == y</code>	如果表达式 <code>x</code> 的值等于表达式 <code>y</code> 的值，则返回 <code>true</code>
<code>x != y</code>	如果表达式 <code>x</code> 的值不等于表达式 <code>y</code> 的值，则返回 <code>true</code>
<code>x >= y</code>	如果表达式 <code>x</code> 的值大于等于表达式 <code>y</code> 的值，则返回 <code>true</code>
<code>x > y</code>	如果表达式 <code>x</code> 的值大于表达式 <code>y</code> 的值，则返回 <code>true</code>

布尔值和表达式可以用于if语句中，也可以用于for语句的条件中，以及switch语句的case子句的条件判断中，这些都将在第5章讲述。

二元逻辑操作符（`||`和`&&`）使用短路逻辑。这意味着如果我们的表达式是`b1||b2`，并且表达式`b1`的值为`true`，那么无论`b2`的值为什么，表达式的结果都为`true`，因此`b2`的值不会再计算而直接返回`true`。类似地，如果我们的表达式为`b1&& b2`，而表达式`b1`的计算结果为`false`，那

么无论表达式**b2**的值是什么，都不会再计算它的值，而直接返回 **false**。

Go语言会严格筛选用于使用比较操作符（**<**、**<=**、**==**、**!=**、**>=**、**>**）进行比较的值。这两个值必须是相同类型的，或者如果它们是接口，就必须实现了相同的接口类型。如果有一个值是常量，那么它的类型必须与另一个类型相兼容。这意味着一个无类型的数值常量可以跟另一个任意数值类型的值进行比较，但是不同类型且非常量的数值不能直接比较，除非其中一个被显式的转换成与另一个相同类型的值。（数字之间转换的内容已在2.3节讨论过。）

==和**!=**操作符可以用于任何可比较的类型，包括数组和结构体，只要它们的元素和成员变量与**==**和**!=**操作符相兼容。这些操作符不能用于比较切片，尽管这种比较可以通过 **Go** 标准库中的 **reflect.DeepEqual()**函数来完成。**==**和**!=**操作符可以用于比较两个指针和接口，或者将指针、接口或者引用（比如指向通道、映射或切片）与**nil**比较。别的比较操作符（**<**、**<=**、**>=**和**>**）只适用于数字和字符串。（由于**Go**也跟**C**和**Java**一样，不支持操作符重载，对于我们自定义的类型，如果需要，可以实现自己的比较方法或者函数，如**Less()**或者**Equal()**，详见第6章。）

2.3 数值类型

Go语言提供了大量内置的数值类型，标准库也提供了**big.Int**类型的整数和**big.Rat**类型的有理数，这些都是大小不限的（只限于机器的内存）。每一个数值类型都不同，这意味着我们不能在不同的类型（例如，类型**int32**和类型**int**）之间进行二进制数值运算或者比较操作（如**+**或者**<**）。无类型的数值常量可以兼容表达式中任何（内置的）类型

的数值，因此我们可以直接将一个无类型的数值常量与另一个数值做加法，或者将一个无类型的常量与另一个数值进行比较，无论另一个数值是什么类型（但必须为内置类型）。

如果我们需要在不同的数值类型之间进行数值运算或者比较操作，就必须进行类型转换，通常是将类型转换成最大的类型以防止精度丢失。类型转换采用 `type(value)` 的形式，只要合法，就总能转换成功——即使会导致数据丢失。请看下面的例子。

```
const factor = 3           // factor与任何数值类型兼容
i := 20000                 // 通过推断得出i的类型为int
i *= factor
j := int16(20)             // j的类型为int16，与这样定义效果一
样: var j int16 = 20
i += int(j)                // 类型必须匹配，因此需要转换
k := uint8(0)              // 效果与这样定义一样: var k uint8
k = uint8(i)               // 转换成功，但是k的值被截为8位
fmt.Println(i, j, k)       // 打印: 60020 20 16
```

为了执行缩小尺寸的类型转换，我们可以创建合适的函数。例如：

```
func Uint8FromInt(x int) (uint8, error) {
    if 0 <= x && x <= math.MaxUint8 {
        return uint8(x), nil
    }
    return 0, fmt.Errorf("%d is out of the uint8 range", x)
}
```

该函数接受一个 `int` 型参数，如果给定的 `int` 值在给定的范围内，则返回一个 `uint8` 和 `nil`，否则返回 `0` 和相应的错误值。`math.MaxUint8` 常量来自于 `math` 包，该包中也有一些类似的 Go 语言中其他内置类型的常

量。（当然，无符号的类型没有最小值常量，因为它们的最小值都为0。）`fmt.Errorf()`函数返回一个基于给定的格式化字符串和值创建的错误值。（字符串格式化的内容将在3.5节讨论。）

相同类型的数值可以使用比较操作符进行比较（参见表 2-3）。类似地，Go语言的算术操作符可以用于数值。表 2-4 给出的算术运算操作符可用于任何内置的数值，而表 2-6 给出的算术运算操作符适用于任何整型值。

表2-4 可用于任何内置的数值的算术运算操作符

语法	描述/结果
<code>+x</code>	<code>x</code>
<code>-x</code>	<code>x</code> 的负值
<code>x++</code>	为 <code>x</code> 加上一个无类型的常量 1
<code>x--</code>	为 <code>x</code> 减去一个无类型的常量 1
<code>x += y</code>	将 <code>x</code> 加上 <code>y</code>
<code>x -= y</code>	将 <code>x</code> 减去 <code>y</code>
<code>x *= y</code>	将 <code>x</code> 乘以 <code>y</code>
<code>x /= y</code>	将 <code>x</code> 除以 <code>y</code> ，如果这些数字都是整数那么任何余数都被丢弃，除以 0 会导致运行时异常
<code>x + y</code>	<code>x</code> 与 <code>y</code> 的和
<code>x - y</code>	<code>x</code> 减去 <code>y</code> 的结果
<code>x * y</code>	<code>x</code> 乘以 <code>y</code> 的结果
<code>x / y</code>	<code>x</code> 除以 <code>y</code> 的结果，如果这些数字都是整数那么任何余数都被丢弃，除以 0 会导致运行时异常 ^①

① 异常，即panic，见1.6节和5.5节。

常量表达式的值在编译时计算，它们可能使用任何算术、布尔以及比较操作符。例如：

```
const (  
    efri int64 = 100000000000 // 类型: int64  
    hlutföllum = 16.0 / 9.0 // 类型: float64  
    mælikvarða = complex(-2, 3.5) * hlutföllum // 类型: complex128  
    erGjaldgengur = 0.0 <= hlutföllum && hlutföllum < 2.0 // 类型:  
    bool
```

)

该例子使用冰岛语标识符表示 Go语言完全支持本土语言的标识符。（我们马上会讨论`complex()`，参见2.3.2节。）

虽然Go语言的优先级规则比较合理（即不像C和C++那样），我们还是推荐使用括号来保证清晰的含义。强烈推荐使用多种语言进行编程的程序员使用括号，以避免犯一些难以发现的错误。

2.3.1 整型

Go语言提供了11种整型，包括5种有符号的和5种无符号的，再加上1种用于存储指针的整型类型。它们的名字和值在表2-5中给出。另外，Go语言允许使用`byte`来作为无符号`uint8`类型的同义词，并且使用单个字符（即Unicode码点）的时候提倡使用`rune`来代替`int32`。大多数情况下，我们只需要一种整型，即`int`。它可以用于循环计数器、数组和切片索引，以及任何通用目的的整型运算符。通常，该类型的处理速度也是最快的。本书撰写时，`int`类型表示成一个有符号的32位整型（即使在64位平台上也是这样的），但在Go语言的新版本中可能会改成64位的。

表2-5 Go语言的整数类型及其范围

类型	取值范围
byte	等同于 uint8
int	依赖不同平台下的实现，可以是 int32 或者 int64
int8	[-128, 127]
int16	[-32 768, 32 767]
int32	[-2 147 483 648, 2 147 483 647]
int64	[-9 223 372 036 854 775 808, 9 223 372 036 854 775 807]
rune	等同于 uint32
uint	依赖不同平台下的实现，可以是 uint32 或者 uint64
uint8	[0, 255]
uint16	[0, 65 535]
uint32	[0, 4 294 967 295]
uint64	[0, 18 446 744 073 709 551 615]
uintptr	一个可以恰好容纳指针值的无符号整数类型（对 32 位平台是 uint32，对 64 位平台是 uint64）

从外部程序（如从文件或者网络连接）读写整数时，可能需要别的整数类型。这种情况下需要确切地知道需要读写多少位，以便处理该整数时不会发生错乱。

常用的做法是将一个整数以 `int` 类型存储在内存中，然后在读写该整数的时候将该值显式地转换为有符号的固定尺寸的整数类型。`byte(uint8)` 类型用于读或者写原始的字节。例如，用于处理 UTF-8 编码的文本。在前一章的 `americanise` 示例中，我们讨论了读写 UTF-8 编码的文本的基本方式，第 8 章中我们会继续讲解如何读写内置以及自定义的数据类型。

Go 语言的整型支持表 2-4 中所列的所有算术运算，同时它们也支持表 2-6 中所列出的算术和位运算。所有这些操作的行为都是可预期的，特别是本书给出了很多示例，因此无需更深入讨论。

表 2-6 只适用于内置的整数类型的算术运算操作符

语法	含义/结果
<code>^x</code>	按位取反
<code>x %= y</code>	将 <code>x</code> 的值设为 <code>x</code> 除以 <code>y</code> 的余数；除 0 会导致一个运行时异常
<code>x &= y</code>	将 <code>x</code> 的值设为 <code>x</code> 和 <code>y</code> 按位与（AND）的结果
<code>x = y</code>	将 <code>x</code> 的值设为 <code>x</code> 和 <code>y</code> 按位或（OR）的结果
<code>x ^= y</code>	将 <code>x</code> 的值设为 <code>x</code> 和 <code>y</code> 按位异或（XOR）的结果
<code>x &^= y</code>	将 <code>x</code> 的值设为 <code>x</code> 和 <code>y</code> 按位与非（ANDNOT）的结果
<code>x >>= u</code>	将 <code>x</code> 的值设为 <code>x</code> 右移 <code>u</code> 个位的结果
<code>x <<= u</code>	将 <code>x</code> 的值设为 <code>x</code> 左移 <code>u</code> 个位的结果
<code>x % y</code>	结果为 <code>x</code> 除以 <code>y</code> 的余数
<code>x & y</code>	结果为 <code>x</code> 和 <code>y</code> 按位与（AND）
<code>x y</code>	结果为 <code>x</code> 和 <code>y</code> 按位或（OR）
<code>x ^ y</code>	结果为 <code>x</code> 和 <code>y</code> 按位异或（XOR）
<code>x &^ y</code>	结果为 <code>x</code> 和 <code>y</code> 按位与非（ANDNOT）
<code>x << u</code>	结果为 <code>x</code> 左移 <code>u</code> 个位
<code>x >> u</code>	结果为 <code>x</code> 右移 <code>u</code> 个位

将一个更小类型的整数转换成一个更大类型的整数总是安全的（例如，从 `int16` 转换成 `int32`），但是如果向下转换一个太大的整数到一个目标类型或者将一个负整数转换成一个无符号整数，则会产生无声的截断或者一个不可预期的值。这种情况下最好使用一个自定义的向下转换函数，如前文给出的那个。当然，当试图向下转换一个字面量时（如 `int8(200)`），编译器会检测到问题，并报告异常错误。也可以使用标准 Go 语法将整数转换成浮点型数字（如 `float64(integer)`）。

有些情况下，Go 语言对 64 位整数的支持让使用大规格的整数来进行高精度计算成为可能。例如，在商业上计算财务时使用 `int64` 类型的整数来表示百万分之一美分，可以使得在数十亿美元之内计算还保持着足够高的精度，这样做有很多用途，特别是当我们很关心除法操作的时候。如果计算财务时需要完美的精度，并且需要避免余数错误，我们可以使用 `big.Rat` 类型。

大整数

有时我们需要使用甚至超过int64位和uint64位的数字进行完美的计算。这种情况下，我们就不能使用浮点数了，因为它们表示的是近似值。幸运的是，Go语言的标准库提供了两个无限精度的整数类型：用于整数的big.Int型以及用于有理数的big.Rat型（即包括可以表示成分数

$\frac{2}{3}$

的数字如 $\frac{2}{3}$ 和1.1496，但不包括无理数如e或者 π ）。这些整数类型可以保存任意数量的数字——只要机器内存足够大，但是其处理速度远比内置的整型慢。

Go语言也像C和Java一样不支持操作符重载，提供给big.Int和big.Rat类型的方法有它自己的名字，如Add()和Mul()。在大多数情况下，方法会修改它们的接收器（即调用它们的大整数），同时会返回该接收器来支持链式操作。我们并没有列出math/big包中提供的所有函数和方法，它们都可以在文档上查到，并且也可能在本书出版之后又添加了新内容。但是，我们会给出一个具有代表性的例子来看看big.Int是如何使用的。

使用Go语言内置的float64类型，我们可以很精确地计算包含大约15位小数的情况，这在大多数情况下足够了。但是，如果我们想要计算包含更多位小数，即数十个甚至上百个小数时，例如计算 π 的时候，那么就没有内置的类型可以满足。

1706年，约翰·梅钦（John Machin）发明了一个计算任意精度 π 值的公式（见图2-1），我们可以将该公式与Go标准库中的big.Int结合起来计算 π ，以得到任意位数的值。在图2-1中给出了该公式以及它依赖的arccot()函数。（理解这里介绍的big.Int包的使用无需理解梅钦的公式。）我们实现的arccot()函数接受一个额外的参数来限制计算结果的精度，以防止超出所需的小数位。

$$\pi = 4 \times (4 \times \operatorname{arccot}(5) - \operatorname{arccot}(239)) \quad \left| \quad \operatorname{arccot}(x) = \frac{1}{x} - \frac{1}{3x^3} + \frac{1}{5x^5} - \frac{1}{7x^7} + \dots$$

图2-1 Machin的公式

整个程序在文件`pi_by_digits/pi_by_digits.go`中，不到80行。下面是它的`main()`函数 [1] 。

```
func main() {
    places := handleCommandLine(1000)
    scaledPi := fmt.Sprint( $\pi$ (places))
    fmt.Printf("3.%.5s\n", scaledPi[1:])
}
```

该程序假设默认的小数位数为1 000，但是用户可以在命令行中指定任意的小数位数。`handleCommandLine()`函数（这里没有给出）返回传递给它的值，或者是用户从命令行输入的数字（如果有并且是合法的话）。 `π ()`函数将 π 以`big.Int`型返回，它的值为314159...。我们将该值打印到一个字符串，然后将字符串以适当的格式打印到终端，以便看起来像3.1415926535897 9323846264338327950288419716939937510 这样（这里我们打印了将近50位）。

```
func  $\pi$ (places int) *big.Int {
    digits := big.NewInt(int64(places))
    unity := big.NewInt(0)
    ten := big.NewInt(10)
    exponent := big.NewInt(0)
    unity.Exp(ten, exponent.Add(digits, ten), nil) ①
    pi := big.NewInt(4)
    left := arccot(big.NewInt(5), unity)
    left.Mul(left, big.NewInt(4)) ②
```

```

    right := arccot(big.NewInt(239), unity)
    left.Sub(left, right)
    pi.Mul(pi, left) ③
    return pi.Div(pi, big.NewInt(0).Exp(ten, ten, nil)) ④
}

```

$\pi()$ 函数开始时计算unity变量的值 ($10^{\text{digits}+10}$)，我们将其当做一个放大因子来使用，以便计算的时候可以使用整数。为了防止余数错误，使用+10操作为用户添加额外10个数字。然后，我们使用了梅钦公式，以及我们修改过的接受unity变量作为其第二个参数的arccot()函数（没有给出）。最后，我们返回除以 10^{10} 的结果，以还原放大因子unity的效果。

为了让unity变量保存正确的值，我们开始创建4个变量，它们的类型都是*big.Int（即指向big.Int的指针，参见4.1节）。unity和exponent变量都被初始化成0，变量ten初始化成10，digits被初始化成用户请求的数字的位数。unity值的计算一行就完成了（①）。big.Int.Add()方法往变量digits中添加了10。然后big.Int.Exp()方法用于将10增大到它的第二个参数（digits+10）的幂。如果第三个参数像这里一样是nil，big.Int.Exp(x, y, nil)进行 x^y 计算。如果3个参数都是非空的，big.Int.Exp(x, y, z)执行（ x^y 模z）。值得注意的是，我们无需将结果赋给unity变量，这是因为大部分big.Int方法返回的同时会修改它的接收器，因此在这里unity被修改成包含结果值。

接下来的计算模式类似。我们为pi设置一个初始值4，然后返回梅钦公式内部的左半部分。创建完成之后，我们无需将left的值赋回去（②），因为big.Int.Mul()方法会在返回时将结果（我们可以安全地忽略它）保存回其接收器中（在本例中即保存回left变量中）。接下来，我们计算公式内部右半部分的值，并从left中减去right的值（将其结果

保存在`left`）中。现在我们用`pi`（其值为4）乘以`left`（它保存了梅钦公式的结果）。这样就得到了结果，只是被放大了`unity`倍。因此，在最后一行中（④），我们将其值除以（ 10^{10} ）以还原其结果。

使用`big.Int`类型需小心，因为它的大多数方法都会修改它的接收器（这样做是为了节省创建大量临时`big.Int`值的开销）。与执行`pi×left`计算并将计算结果保存在`pi`中的那一行（③）相比，我们计算`pi÷1010`并将结果立即返回（④），而无需关心`pi`的值最后已经被修改。

无论什么时候，最好只使用`int`类型，如果`int`型不能满足则使用`int64`型，或者如果不是特别关心它们的近似值，则可以使用`float32`或者`float64`类型。然而，如果计算需要完美的精度，并且我们愿意付出使用内存和处理器的代价，那么就使用 `big.Int` 或者 `big.Rat`类型。后者在处理财务计算时特别有用。进行浮点计算时，如果需要可以像这里所做的那样对数值进行放大。

2.3.2 浮点类型

Go语言提供了两种类型的浮点类型和两种类型的复数类型，它们的名称及相应的范围在表 2-7中给出。浮点型数字在 Go语言中以广泛使用的 IEEE-754 格式表示（http://en.wikipedia.org/wiki/IEEE_754-2008）。该格式也是很多处理器以及浮点数单元所使用的原生格式，因此大多数情况下Go语言能够充分利用硬件对浮点数的支持。

表2-7 Go语言的浮点类型

类型	范围
float32	$\pm 3.402\ 823\ 466\ 385\ 288\ 598\ 117\ 041\ 834\ 845\ 169\ 254\ 40 \times 10^{38}$ 尾数部分计算精度大概是 7 个十进制数
float64	$\pm 1.797\ 693\ 134\ 862\ 315\ 708\ 145\ 274\ 237\ 317\ 043\ 567\ 981 \times 10^{308}$ 尾数部分计算精度大概是 15 个十进制数
complex64	实部和虚部都是一个 float32
complex128	实部和虚部都是一个 float64

Go语言的浮点数支持表2-4中所有的算术运算。math包中的大多数常量以及所有函数都在表2-8和表2-10中列出。

表2-8 math包中的常量与函数 #1

除非特殊说明，math包中的所有函数都接受并且返回 float64 数据。所有给出的常量都截成小数点后面包含 15 位，以更好地适应表。

语法	含义/结果
math.Abs(x)	$ x $ ，即 x 的绝对值
math.Acos(x)	以弧度为单位的 x 的反余弦值
math.Acosh(x)	以弧度为单位的 x 的反双曲余弦值
math.Asin(x)	以弧度为单位的 x 的正弦值
math.Asinh(x)	以弧度为单位的 x 的反双曲正弦值
math.Atan(x)	以弧度为单位的 x 的反正切值
math.Atan2(y, x)	坐标系 x 正方向与射线 (x, y) 构成的角度的反正切值
math.Atanh(x)	以弧度为单位的 x 的反双曲正切值
math.Cbrt(x)	$\sqrt[3]{x}$ ，即 x 的开立方根
math.Ceil(x)	$\lceil x \rceil$ ，即 $\geq x$ 的最小整数值；例如 <code>math.Ceil(5.4) == 6.0</code>

续表

语法	含义/结果
<code>math.Copysign(x, y)</code>	得到一个值，其绝对值与 x 相同，但符号位与 y 相同
<code>math.Cos(x)</code>	以弧度为单位的 x 的余弦值
<code>math.Cosh(x)</code>	以弧度为单位的 x 的双曲余弦值
<code>math.Dim(x, y)</code>	效果上，等价于 <code>math.Max(x - y, 0.0)</code>
<code>math.E</code>	自然数 e ；值大约是 2.718 281 828 459 045
<code>math.Erf(x)</code>	$\text{erf}(x)$ ，即 x 的高斯误差函数
<code>math.Erfc(x)</code>	$\text{erfc}(x)$ ，即 x 的互补高斯误差函数
<code>math.Exp(x)</code>	即 e^x
<code>math.Exp2(x)</code>	即 2^x
<code>math.Expm1(x)</code>	即 $e^x - 1$ ；但当 x 接近于 0 时，其结果的精度远好于用 <code>math.Exp(x) - 1</code>
<code>math.Float32bits(f)</code>	依据 IEEE-754 标准表示的 float32 值，并将其视为 int32 整数
<code>math.Float32frombits(u)</code>	是上面 <code>math.Float32bits(f)</code> 的反操作，将一个 int32 整数视作符合 IEEE-754 标准表示的 float32
<code>math.Float64bits(x)</code>	依据 IEEE-754 标准表示的 float64 值，并将其视为 uint64 整数
<code>math.Float64frombits(u)</code>	是上面 <code>math.Float64bits(x)</code> 的反操作，将一个 uint64 整数视作符合 IEEE-754 标准表示的 float64

表2-9 math包中的常量与函数 #2

语法	含义/结果
<code>math.Floor(x)</code>	$\lfloor x \rfloor$ ，即 $\leq x$ 的最大整数值；例如 <code>math.Floor(5.4) == 5.0</code>
<code>math.Frexp(x)</code>	结果是 <code>(frac float64, exp int)</code> ，使得 $x = \text{frac} * 2^{\text{exp}}$ ；是 <code>math.Ldexp(frac, exp)</code> 的反函数
<code>math.Gamma(x)</code>	$\Gamma(x)$ ，即 $(x-1)!$
<code>math.Hypot(x, y)</code>	<code>math.Sqrt(x*x + y*y)</code>
<code>math.Ilogb(x)</code>	取 $\log_2 x$ 的整数部分；参见 <code>math.Logb()</code>
<code>math.Inf(n)</code>	如果 $n \geq 0$ ，则返回 float64 类型的 $+\infty$ 值；否则返回 $-\infty$
<code>math.IsInf(x, n)</code>	如果 $n > 0$ 且 x 是 float64 类型的 $+\infty$ 值，或者 $x < 0$ 且 x 是 float64 类型的 $-\infty$ 值，或者 $x == 0$ 且 x 是 float64 类型的 $+\infty$ 或 $-\infty$ 值，则返回 true；否则返回 false
<code>math.IsNaN(x)</code>	如果 x 是 IEEE-754 中的 NaN(not a number)，返回 true；否则返回 false
<code>math.J0(x)</code>	$J_0(x)$ ，第一类贝塞尔函数
<code>math.J1(x)</code>	$J_1(x)$ ，第一类贝塞尔函数
<code>math.Jn(n, x)</code>	$J_n(x)$ ，第一类贝塞尔函数
<code>math.Ldexp(x, n)</code>	结果为 $x2^n$ ，是 <code>math.Frexp</code> 的反函数

续表

语法	含义/结果
<code>math.Lgamma(x)</code>	结果是(<code>lgamma float64, sign int</code>), 使得 $\text{lgamma} = \ln(\Gamma(x))$, $\text{sign} = \Gamma(x)$ 的符号位 (小于 0 时取 -1, 否则取 +1)
<code>math.Ln2</code>	常数 $\ln 2$, 近似等于 0.693 147 180 559 945
<code>math.Ln10</code>	常数 $\ln 10$, 近似等于 2.302 585 092 994 045
<code>math.Log(x)</code>	$\ln x$
<code>math.Log2E</code>	常数 $\frac{1}{\ln 2}$, 近似等于 1.442 695 021 629 333
<code>math.Log10(x)</code>	$\log_{10} x$
<code>math.Log10E</code>	常数 $\frac{1}{\ln 10}$, 近似等于 0.434 294 492 006 301
<code>math.Log1p(x)</code>	$\ln(1+x)$, 但当 x 接近于 0 时, 其结果的精度远好于用 <code>math.Log(1+x)</code>
<code>math.Log2(x)</code>	$\log_2 x$
<code>math.Logb(x)</code>	取 $\log_2 x$ 的整数部分; 参见 <code>math.Ilogb()</code>
<code>math.Max(x, y)</code>	取 x 和 y 中的大者
<code>math.Min(x, y)</code>	取 x 和 y 中的小者
<code>math.Mod(x, y)</code>	取 x 除以 y 的余数; 参见 <code>math.Remainder()</code>

表2-10 math包中的常量与函数 #3

语法	含义/结果
<code>math.Modf(x)</code>	结果是(<code>whole float64, frac float64</code>), 其中 <code>whole</code> = x 的整数部分, 而 <code>frac</code> 是分数部分
<code>math.NaN(x)</code>	返回 IEEE-754 中的 NaN 值
<code>math.Nextafter(x, y)</code>	返回 x 向 y 的下一个可表达的值(译者注:此函数可用于实现 <code>for x != y { ...; x = math.Nextafter(x, y) }</code> 这样的循环)
<code>math.Pi</code>	常数 π , 近似等于 3.141 592 653 589 793
<code>math.Phi</code>	常数 ϕ , 近似等于 1.618 033 988 749 984
<code>math.Pow(x, y)</code>	x^y
<code>math.Pow10(n)</code>	10^n
<code>math.Remainder(x, y)</code>	与 IEEE-754 兼容的 x 除以 y 的余数; 参见 <code>math.Mod()</code>
<code>math.Signbit(x)</code>	如果 $x < 0$ 则返回 <code>true</code>
<code>math.Sin(x)</code>	以弧度为单位的 x 的正弦值
<code>math.SinCos(x)</code>	这个函数主要同时返回 $\sin(x)$ 和 $\cos(x)$
<code>math.Sinh(x)</code>	以弧度为单位的 x 的双曲正弦值
<code>math.Sqrt(x)</code>	\sqrt{x}

续表

语法	含义/结果
<code>math.Sqrt2</code>	常数 $\sqrt{2}$ ，近似等于 1.414213562373095
<code>math.SqrtE</code>	常数 \sqrt{e} ；近似等于 1.648 721 270 700 128
<code>math.SqrtPi</code>	常数 $\sqrt{\pi}$ ；近似等于 1.772 453 850 905 516
<code>math.SqrtPhi</code>	常数 $\sqrt{\phi}$ ；近似等于 1.272 019 649 514 068
<code>math.Tan(x)</code>	以弧度为单位的 x 的正切值
<code>math.Tanh(x)</code>	以弧度为单位的 x 的双曲正切值
<code>math.Trunc(x)</code>	将 x 的分数部分设为 0
<code>math.Y0(x)</code>	$Y_0(x)$ ，第二类贝塞尔函数
<code>math.Y1(x)</code>	$Y_1(x)$ ，第二类贝塞尔函数
<code>math.Yn(n, x)</code>	$Y_n(x)$ ，第二类贝塞尔函数

浮点型数据使用小数点的形式或者指数符号来表示，例如0.0、3.、8.2、-7.4、-6e4、.1以及5.9E-3等。计算机通常使用二进制表示浮点数，这意味着有些小数可以精确地表示（如0.5），但是其他的浮点数就只能近似表示（如 0.1和0.2）。另外，这种表示使用固定长度的位，因此它所能表示的数字的位数有限。这不是Go语言特有的问题，而是困扰所有主流语言的浮点数问题。然而，这种不精确性并不是总都这么明显，因为 Go语言使用了智能算法来输出浮点数，这些浮点数在保证精确性的前提下使用尽可能少的数字。

表2-3中所列出的所有比较操作都可以用于浮点数。不幸的是，由于浮点数是以近似值表示的，用它们来做相等或者不相等比较时并不总能得到预期的结果。

```

x, y := 0.0, 0.0
for i := 0; i < 10; i++ {
    x += 0.1
    if i%2 == 0{
        y += 0.2
    } else {
        fmt.Printf("%-5t %-5t %-5t %-5t", x == y, EqualFloat(x, y, -1),

```

```

        EqualFloat(x, y, 0.0000000000000001), EqualFloatPrec(x, y, 6))
    fmt.Println(x, y)
}
}

```

```

true true true true 0.2 0.2
true true true true 0.4 0.4
false false true true 0.6 0.60000000000000000001
false false true true 0.79999999999999999999 0.8
false false true true 0.99999999999999999999 1

```

这里开始时我们定义了两个float64型的浮点数，其初始值都为0。我们往第一个值中加上10个0.1，往第二个值中加上5个0.2，因此结果都为1。然而，正如代码片段下面所给出的输出所示，有些浮点数并不能得到完美的结果。这样看来，计算使用==以及!=对浮点数进行比较时，我们必须非常小心。当然，有些情况下可以使用内置的操作符来比较浮点数的相等或者不相等性。例如，为了避免除数为0，可以这样做if y != 0.0 { return x / y}。

格式 "%-5 " 以一个向左对齐的5个字符宽的区域打印一个布尔值。字符串格式化的内容将在下一章讲解，参见3.5节。

```

func EqualFloat(x, y, limit float64) bool {
    if limit <= 0.0 {
        limit = math.SmallestNonzeroFloat64
    }
    return math.Abs(x-y) <= (limit * math.Min(math.Abs(x),
math.Abs(y)))
}

```

EqualFloat()函数用于在给定精度范围内比较两个float64型数，如果给定的精度范围为负数（如-1），则将该精度设为机器所能达到的

最大精度。它还依赖于标准库`math`包中的一个函数（以及一个常量）。

一个可替代（也更慢）的方式是以字符串的形式比较两个数字。

```
func EqualFloatPrec(x, y float64, decimals int) bool {  
    a := fmt.Sprintf("%.*f", decimals, x)  
    b := fmt.Sprintf("%.*f", decimals, y)  
    return len(a) == len(b) && a == b  
}
```

对于该函数，其精度以小数点后面数字的位数声明。`fmt.Sprintf()`函数的`%`格式化参数能够接受一个`*`占位符，用于输入一个数字，因此这里我们基于给定的`float64`创建了两个字符串，每个字符串都以给定位数的尾数进行格式化。如果浮点数中数字的多少不一样，那么字符串`a`和`b`的长度也不一样（例如，`12.32`和`592.85`），这样就能给我们一个快速的短路测试。（字符串格式化的内容将在3.5节讲解。）

大多数情况下如果需要浮点数，`float64`类型是最好的选择，一个特别原因是`math`包中的所有函数都使用`float64`类型。然而，Go语言也支持`float32`类型，这在内存比较宝贵并且无需使用`math`包，或者愿意处理在与`float64`类型之间进行来回转换的不便时非常有用。由于Go语言的浮点类型是固定长度的，因此从外部文件或者网络连接中读写时非常安全。

使用标准的Go语法（例如`int(float)`）可以将浮点型数字转换成整数，这种情况下小数部分会被丢弃。当然，如果浮点数的值超出了目标整型的范围，那么得到的结果值将是不可预期的。我们可以使用一个安全的转换函数来解决该问题。例如：

```
func IntFromFloat64(x float64) int {  
    if math.MinInt32 <= x && x <= math.MaxInt32 {  
        whole, fraction := math.Modf(x)
```

```

    if fraction >= 0.5 {
        whole++
    }
    return int(whole)
}

panic(fmt.Sprintf("%g is out of the int32 range", x))
}

```

Go语言规范（golang.org/doc/go_spec.html）中说明了int型所占的位数与uint相同，并且uint总是32位或者64位的。这意味着一个int型值至少是32位的，我们可以安全地使用math.MinInt32和math.MaxInt32常量来作为int的范围。

我们使用 math.Modf()函数来分离给定数字（都是 float64 型数字）的整数以及分数部分，而非简单地返回整数部分（即截断），如果小数部分大于或者等于0.5，则向上取整。

与我们的自定义 Uint8FromInt()函数不同的是，我们不是返回一个错误值，而是将值越界当做一个需要停止程序运行的重要问题，因此我们使用了内置的panic()函数，它会产生一个运行时异常，并停止程序继续运行，直到该异常被一个recover()调用恢复（参见5.5节）。这意味着如果程序运行成功，我们就知道转换过程没有发生值越界。（值得注意的是，该函数并没有以一个return语句结束，Go编译器足够智能，能够意识到panic()调用意味那里不会出现正常的返回值。）

复数类型

Go语言支持的两种复数类型已在表2-7中给出。复数可以使用内置的complex()函数或者包含虚部数值的常量来创建。复数的各部分可以使用内置的real()和imag()函数来获得，这两个函数返回的都是float64型数（或者对于complex64类型的复数，返回一个float32型数）。

复数支持表2-4中所有的算术操作符。唯一可用于复数的比较操作符是==和!=（参见表2-3），但也会遇到与浮点数比较相同的问题。标准库中有一个复数包math/cmplx，表2-11给出了它的函数。

表2-11 Complex数学包中的函数
导入 "math/cmplx" 包。除非特别说明，否则所有的函数都接收和返回 complex128 值。

语法	含义/结果
cmplx.Abs(x)	x ，即作为 float64 值的 x 的绝对值
cmplx.Acos(x)	x 的反余弦，单位为弧度
cmplx.Acosh(x)	x 的反双曲余弦，单位为弧度
cmplx.Asin(x)	x 的正弦，单位为弧度
cmplx.Asinh(x)	x 的反双曲正弦，单位为弧度
cmplx.Atan(x)	x 的正切，单位为弧度
cmplx.Atanh(x)	x 的反双曲正切，单位为弧度
cmplx.Conj(x)	x 的复共轭
cmplx.Cos(x)	x 的余弦，单位为弧度

续表

语法	含义/结果
<code>cmplx.Cosh(x)</code>	x 的双曲余弦，单位为弧度
<code>cmplx.Cot(x)</code>	x 的余切，单位为弧度
<code>cmplx.Exp(x)</code>	e^x
<code>cmplx.Inf()</code>	复数 <code>complex(math.Inf(1), math.Inf(1))</code>
<code>cmplx.IsInf(x)</code>	如果 <code>real(x)</code> 或者 <code>imag(x)</code> 的结果为 $\pm\infty$ ，则为 <code>true</code> ；否则为 <code>false</code>
<code>cmplx.IsNaN(x)</code>	如果 <code>real(x)</code> 或者 <code>imag(x)</code> 都为“非数字”并且都不是 $\pm\infty$ ，则为 <code>true</code> ；否则为 <code>false</code>
<code>cmplx.Log(x)</code>	$\ln x$
<code>cmplx.Log10(x)</code>	$\lg x$
<code>cmplx.NaN()</code>	复数“非数字”的值
<code>cmplx.Phase(x)</code>	<code>float64</code> 型数字 x 在范围 $[-\pi, +\pi]$ 内的相
<code>cmplx.Polar(x)</code>	求满足以下等式的 r 与 θ 值，均为 <code>float64</code> ，其中相 r 的范围为 $[-\pi, +\pi]$
<code>cmplx.Pow(x, y)</code>	x^y
<code>cmplx.Rect(r, θ)</code>	坐标为 r ，相为 θ 构成的 <code>complex128</code> 复数
<code>cmplx.Sin(x)</code>	x 的正弦值，单位为弧度
<code>cmplx.Sinh(x)</code>	x 的双曲正弦值，单位为弧度
<code>cmplx.Sqrt(x)</code>	\sqrt{x}
<code>cmplx.Tan(x)</code>	x 的正切，单位为弧度
<code>cmplx.Tanh(x)</code>	x 的双曲正切，单位为弧度

这里有些简单的例子：

`f := 3.2e5` // 类型： `float64`

`x := -7.3 - 8.9i` // 类型： `complex128`（字面量）

`y := complex64(-18.3 + 8.9i)` // 类型： `complex64`（转换）①

`z := complex(f, 13.2)` // 类型： `complex128`（构造）

②

`fmt.Println(x, real(y), imag(z))` // 打印： `(-7.3-8.9i) -18.3 13.2`

正如数学中所表示的那样，Go语言使用后缀*i*表示虚数 [2]。这里，数*x*和*z*都是`complex128`类型的，因此它们的实部和虚部都是`float64`类型的。y是`complex64`类型的，因此它的各部分都是`float32`类型的。需要注意的一点小细节是，使用`complex64`类型的名字（或者是任何其他

内置的类型名) 来作为函数会进行类型转换。因此这里 (①) 复数 `-18.3+8.9i` (从复数字面量推断出来的复数类型为`complex128`) 被转换成一个`complex64`类型的复数。然而, `complex()`是一个函数, 它接受两个浮点数输入, 返回对应的`complex128` (②) 。

另一个细节点是`fmt.Println()`函数可以统一打印复数。(就像将在第6章看到的那样, 我们可以创建自己的无缝兼容 Go语言的打印函数的类型, 只需为它们简单地添加一个`String()`方法即可实现。)

一般而言, 最适合使用的复数类型是`complex128`, 因为`math/cmplx`包中的所有函数都工作于`complex128`类型。Go语言也支持`complex64`类型, 这在内存非常紧缺的情况下是非常有用的。Go语言的复数类型是定长的, 因此从外部文件或网络连接中读写复数总是安全的。

本章中我们讲解了 Go语言的布尔类型以及数值类型, 同时在表格中给出了可以查询和操作它们的操作符和函数。下一章将讲解Go语言的字符串类型, 包括对Go语言的格式化打印功能(参见 3.5 节)的全面讲解, 当然其中也包括我们需要的格式化打印布尔值和数字的内容。第8章中我们会看看如何对文件进行数据类型的读写, 包括布尔型和数值类型, 在本章结束之前, 我们会讲解一个短小但是完全能够工作的示例程序。

[2.4 例子: statistics](#)

这个例子的目的是为了提高大家对Go编程的理解并提供实践机会。就如同第一章, 这个例子使用了一些还没有完整讲解的Go语言特性。这应该不是大问题, 因为我们提供了相应的简单解释和交叉引用。这个例子还很简单的使用了 Go语言官方网络库 `net/http` 包。使用

net/http包我们可以非常容易地创建一个简单的HTTP服务器。最后，为了不脱离本章的主题，这节的例子和练习都是数值类型的。

statistics程序（源码在statistics/statistics.go文件里）是一个Web应用，先让用户输入一串数字，然后做一些非常简单的统计计算，如图2-2所示。我们分两部分来讲解这个例子，先介绍如何实现程序中相关的数学功能，然后再讲解如何使用net/http包来创建一个Web应用程序。由于篇幅有限，而且书中的源码均可从网上下载，所以有侧重地只显示部分代码（对于import部分和一些常量等可能会被忽略掉），当然，为了让大家能更好地理解我们会尽可能讲解得全面些。

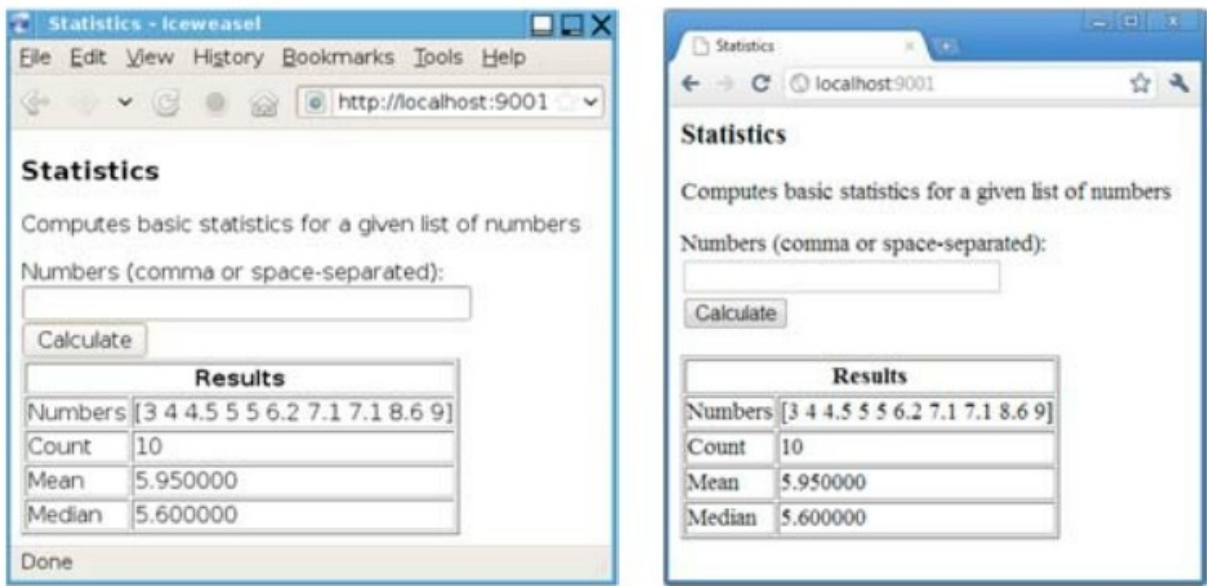


图2-2 Linux和Windows上的Statistics示例程序

2.4.1 实现一个简单的统计函数

我们定义了一个聚合类型的结构体，包含用户输入的数据以及我们准备计算的两种统计：

```
type statistics struct {  
    numbers []float64  
    mean     float64
```

```
    median    float64
}
```

Go语言里的结构体类似于C里的结构体或者Java里只有public数据成员的类（不能有方法），但是不同于C++的结构体，因为它并不是一个类。我们在6.4节将会看到，Go语言里的结构体对聚合和嵌入的支持是非常完美的，是Go语言面向对象编程的核心（主要介绍在第6章）。

```
func getStats(numbers []float64) (stats statistics) {
    stats.numbers = numbers
    sort.Float64s(stats.numbers)
    stats.mean = sum(numbers) / float64(len(numbers))
    stats.median = median(numbers)
    return stats
}
```

`getStats` 函数的作用就是对传入的`[]float64`切片（这些数据都在`processRequest()`里得到）进行统计，然后将相应的结果保存到`stats`结果变量中。其中计算中位数使用了`sort`包里的`Float64s()`函数对原数组进行升序排列（原地排序），也就是说 `getStats()`函数修改了它的参数，这种情况在传切片、引用或者函数指针到函数时是很常见的。如果需要保留原始切片，可以使用Go语言内置的`copy()`函数（参见4.2.3节）将它赋值到一个临时变量，使用临时变量来工作。

结构体中的`mean`（通常也叫平均数）是对一连串的数进行求和然后除以总个数得到的结果。这里我们使用一个辅助函数`sum()`求和，使用内置的`len()`取得切片的大小（总个数）并将其强制转换成`float64`类型的变量（因为`sum()`函数返回一个`float64`的值）。这样我们也就确保了这是一个浮点除法运算，避免了使用整数类型可能带来的精度损失问题。`median`是用来保存中位数的，我们使用`median()`函数来单独计算它。

我们没有检查除数为0的情况，因为在我们的程序逻辑里，`getStats()`函数只有在至少有1个数据的时候才会被调用，否则程序会退出并产生一个运行时异常（`runtime panic`）。对于一个关键性应用当发生一个异常时程序是不应该被结束的，我们可以使用 `recover()`来捕获这个异常，将程序恢复到一个正常的状态，让程序继续运行（5.5节）。

```
func sum(numbers []float64) (total float64) {  
    for _, x := range numbers {  
        total += x  
    }  
    return total  
}
```

这个函数使用一个`for...range`循环遍历一个切片并将所有的数据相加计算出它们的和。Go语言总是将所有变量初始化为0，包括已经命名了的返回变量，例如`total`，这是一个相当有益的设计。

```
func median(numbers []float64) float64 {  
    middle := len(numbers) / 2  
    result := numbers[middle]  
    if len(numbers)%2 == 0 {  
        result = (result + numbers[middle-1]) / 2  
    }  
    return result  
}
```

这个函数必须传入一个已经排序好了的切片，它一开始将切片里最中间的那个数保存到`result`变量中，但是如果总个数是偶数，就会产生两个中间数，我们取这两个中间数的平均值作为中位数返回。

在这一小部分里我们讲解了这个统计程序最主要的几个处理过程，在下一部分我们来看看一个只有简单页面的Web程序的基本实现。

（读者如果对Web编程不感兴趣的话可以略过本节直接跳到练习或者跳到下一章。）

2.4.2 实现一个基本的HTTP服务器

这个statistics程序在本机上提供了一个简单网页，它的主函数如下：

```
func main() {  
    http.HandleFunc("/", homePage)  
    if err := http.ListenAndServe(":9001", nil); err != nil {  
        log.Fatal("failed to start server", err)  
    }  
}
```

http.HandleFunc()函数有两个参数：一个路径，一个当这个路径被请求时会被执行的函数的引用。这个函数的签名必须是func(http.ResponseWriter, *http.Request)我们可以注册多个“路径-函数”对，这里我们只注册了“/”（通常是网页程序的主页）和一个自定义的homePage()函数。

http.ListenAndServe()函数使用给定的TCP地址启动一个Web服务器。这里我们使用localhost和端口9001。如果只指定了端口号而没有指定网络地址，默认情况下网络地址是localhost。当然也可以这样写“localhost:9001”或者“127.0.0.1:9001”。端口的选择是任意的，如果和现有的服务器有冲突的话，比如端口已经被其他进程占用了等，修改代码中的端口为其他端口号即可。http.ListenAndServe()的第二个参数支

持自定义的服务器，为空的话（传一个`nil`参数）表示使用默认的类型。

这个程序使用了一些字符串常量，但是这里我们只展示其中的一个。

```
form = '<form action="/" method="POST">
<label for="numbers">Numbers (comma or space-separated):</label>
<br />
<input type="text" name="numbers" size="30"><br />
<input type="submit" value="Calculate">
</form>'
```

字符串常量`form`包含一个HTML的表单元素，包含一些文本和一个提交按钮。

```
func homePage(writer http.ResponseWriter, request *http.Request) {
    err := request.ParseForm() // 必须在写响应内容之前调用
    fmt.Fprint(writer, pageTop, form)
    if err != nil {
        fmt.Fprintf(writer, anError, err)
    } else {
        if numbers, message, ok := processRequest(request); ok {
            stats := getStats(numbers)
            fmt.Fprint(writer, formatStats(stats))
        } else if message != "" {
            fmt.Fprintf(writer, anError, message)
        }
    }
    fmt.Fprint(writer, pageBottom)
}
```

当统计网站被访问的时候会调用这个函数，`request`参数包含了请求的详细信息，我们可以往`writer`里写入一些响应信息（HTML格式）。

我们从分析这个表单开始吧。这个表单一开始只有一个空的文本输入框（`text`），我们将这个文本输入框标识为“`numbers`”，这样当后面我们处理这个表单的时候就能找到它。表单的`action`设置为“/”，当用户点击`Calculate`按钮的时候这个页面被重新请求了一次。这也就是说不管什么情况这个`homePage()`函数总是会被调用的，所以它必须处理几个情况：没有数据输入、有数据输入或者发生错误了。实际上，所有的工作都是由一个叫`processRequest()`的自定义函数来完成的，它对每一种情况都做了相应的处理。

分析完表单之后，我们将`pageTop`（源码可见）和`form`这两个字符串常量写到`writer`里去（返回数据给客户端），如果分析表单失败我们写入一个错误信息：`anError`是一个格式化字符串，`err`是即将被格式化的`error`值（格式化字符串3.5节会提到）。

```
anError = '<p class="error">%s</p>'
```

如果分析成功了，我们调用自定义函数 `processRequest()`处理用户键入的数据。如果这些数据都是有效的，我们调用之前提到过的`getStats()`函数来计算统计结果，然后将格式化后的结果返回给客户端，如果接受到的数据无效，且我们得到了错误信息，则返回这个错误信息（当这个表单第一次显示的时候是没有数据的，也没有错误发生，这种情况下 `ok` 变量的值是`false`，而且`message`为空）。最后我们打印出`pageBottom`字符串常量（源码可见），用来关闭`<body>`和`<html>`标签。

```
func processRequest(request *http.Request) ([float64, string, bool) {  
    var numbers []float64
```

```

    if slice, found := request.Form["numbers"]; found && len(slice) > 0
    {
        text := strings.Replace(slice[0], ",", " ", -1)
        for _, field := range strings.Fields(text) {
            if x, err := strconv.ParseFloat(field, 64); err != nil {
                return numbers, "" + field + " is invalid", false
            } else {
                numbers = append(numbers, x)
            }
        }
    }
    if len(numbers) == 0 {
        return numbers, "", false // 第一次没有数据被显示
    }
    return numbers, "", true
}

```

这个函数从`request`里读取表单的数据。如果这是用户首次请求的话，表单是空的，“`numbers`”输入框里没有数据，不过这并不是一个错误，所以我们返回一个空的切片、一个空的错误信息和一个`false`布尔型的值，表明从表单里没有读取到任何数据。这些结果将会以空的表单形式被展示出来。如果用户有输入数据的话我们返回一个`[]float64`类型的切片、一个空的错误信息以及`true`；如果存在非法数据，则返回一个可能为空的切片、一个错误消息和`false`。

`request`结构里有一个`map[string][]string`类型的`Form`成员（参见4.3节），它的键是一个字符串，值是一个字符串切片，所以一个键可能有任意多个字符串在它的值里。例如：如果用户键入“5 8.2 7 13 6”，那么这个`Form`里有一个叫“`numbers`”的键，它的值是`[]string{"5 8.2 7 13 6"}`。

6"}，也就是说它的值是一个只有一个字符串的字符串切片（作为对比，这里有一个包含两个字符串的字符串切片：[]string{"1 2 3","a b c"}）。我们检查这个“numbers”键是否存在（应该存在），如果存在，而且它的值至少有一个字符串，那么我们有数据可以读了。

我们使用strings.Replace()函数（第三个参数指明要执行多少次替换，-1表示替换所有）将用户输入中的所有逗号转换为空格，得到一个新的字符串。新字符串里所有数据都是由空格分隔开的，再使用strings.Fields()函数根据空白处将字符串切分成一个字符串切片，这样我们就可以直接使用for...range 循环来遍历它了（strings 这个包的函数参见3.6 节， for...range 循环请参见 5.3 节）。对于每一个字符串，例如“5”、“8.2”等，用strconv.ParseFloat()函数将它转换成float64类型，这个函数需要传入一个字符串和一个位大小如32或者64（参见3.6节）。如果转换失败我们立即返回现有已经转好了的数据切片、一个非空的错误信息和false。如果转换成功我们将转换的结果float64类型的数据追加到numbers切片里去，内置的函数append()可以将一个或多个值和原有切片合并返回一个新的切片，如果原来的切片的容量比长度大的话，这个函数执行的过程是非常快的，效率很高（关于append()参见4.2.3节）。

假如程序没有因为错误退出（存在非法数据），将返回数值和一个空的错误信息以及true。没有数据需要处理（如这个表单第一次被访问的时候）的情况下返回false。

```
func formatStats(stats statistics) string {  
    return fmt.Sprintf('<table border="1">  
    <tr><th colspan="2">Results</th></tr>  
    <tr><td>Numbers</td><td>%v</td></tr>  
    <tr><td>Count</td><td>%d</td></tr>  
    <tr><td>Mean</td><td>%f</td></tr>
```

```
<tr><td>Median</td><td>%f</td></tr>
</table>', stats.numbers, len(stats.numbers), stats.mean, stats.median)
}
```

一旦计算完毕我们必须将结果返回给用户。因为程序是一个Web应用，所以我们需要生成HTML。（Go语言的标准库提供了用于创建数据驱动文本和HTML的text/template和html/template包，但是我们这里的需求比较简单，所以我们选择自己手动写HTML。9.4.2节有一个简单的使用text/template包的例子。）

fmt.Sprintf()是一个字符串格式化函数，需要一个格式化字符串和一个或多个值，将这—个或多个值按照格式中指定的动作（如%v、%d、%f等）进行转换，返回一个新的格式化后的字符串（格式化字符串在3.5节里有非常详细的描述）。我们不需要做任何HTML转义，因为我们所有的值都是数字。（如果需要的话我们可以使用template.HTMLEscape()或者html.EscapeString()函数。）

从这个例子可以了解，假如我们了解基本的HTML语法，使用Go语言来创建一个简单的Web应用是非常容易的。Go语言标准库提供的html、net/http、html/template和text/template等包让整个事情就变得更加简单。

2.5 练习

本章有两道数值相关的练习题。第一题需要修改我们之前的statistics程序。第二题就是动手创建一个Web应用，实现一些简单的数学计算。

（1）复制 statistics 目录为比如 my_statistics，然后修改 my_statistics/statistics.go 代码，实现估算众数和标准差的功能，当用户

点击页面上的Calculate 按钮时能产生类似图2-3所示的结果。

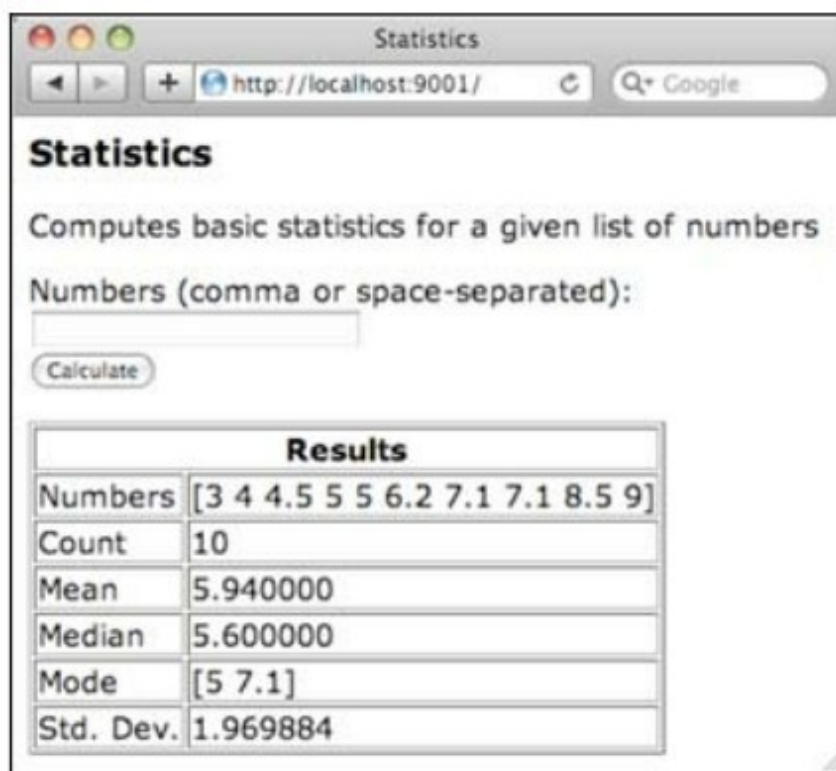


图2-3 MacOSX上的statistics示例程序

这需要在statistics 结构体里增加一些成员并实现两个新函数去执行计算。可以参考statistics_ans/statistics.go 文件里的答案。这大概增加了40 行代码和使用了Go语言内置的append()函数将数字追加到切片里面。

写一个计算标准差的函数也很容易，只需要使用math 包里面的函

$$\sigma = \sqrt{\frac{\sum (x - \bar{x})^2}{n - 1}}$$

数，不到10 行代码就可以完成。我们使用公式来计算，其中x表示每一个数字， \bar{x} 表示数学平均数，n是数字的个数。

众数是指出现最多次的数，可能不止一个，例如有两个或者多个数的出现次数相等。但是，如果所有数的出现次数都是一样的话，我

们就认为众数是不存在的。计算众数要比标准差难，大概需要20行左右的代码。

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

(2) 创建一个Web应用，使用公式 $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ 来求二次方程的解。要用复数，这样即使判别式 $b^2 - 4ac$ 部分为负能计算出方程的解。刚开始的时候可以先让程序能够工作起来，如图2-4左图所示，然后再修改你的代码让它输出得更美观一些，如图2-4右图所示。

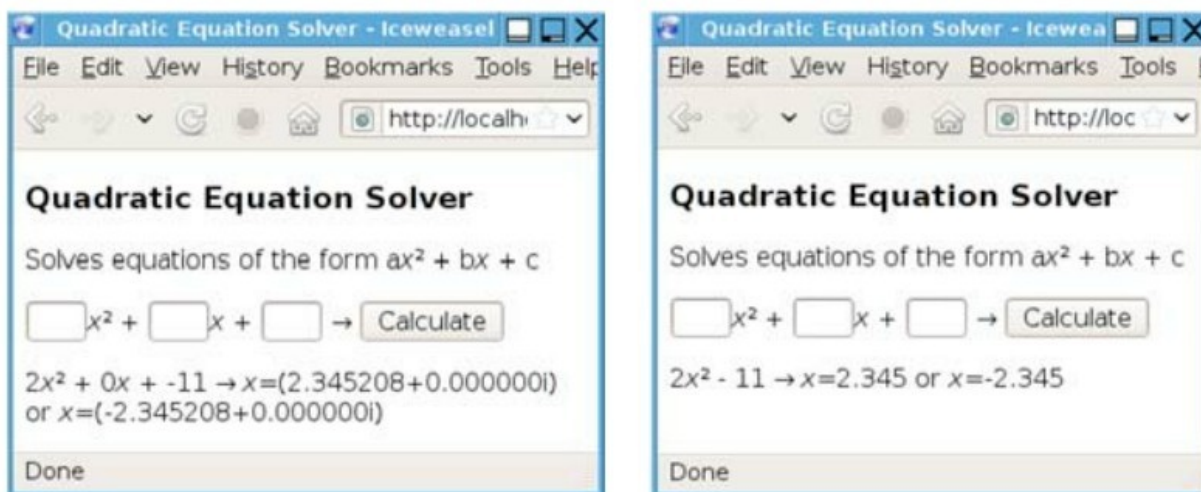


图2-4 Linux上的二次方程求解

最简单的做法就是直接使用 `statistics` 程序的 `main()` 函数、`homePage()` 函数以及 `processRequest()` 函数，然后修改 `homePage()` 让它调用我们自定义的3个函数：`formatQuestion()`、`solve()` 和 `formatSolutions()`，还有 `processRequest()` 函数要用来读取那3个浮点数，这个改动的代码多一点。

第一个参考答案在 `quadratic_ans1/quadratic.go` 里，约120行代码，只实现了基本的功能，使用 `EqualFloat()` 函数来判断方程的两个解是否是约等的，如果约等，只返回一个解。（`EqualFloat()` 函数在之前有讨论过。）

第二个参考答案在`quadratic_ans2/quadratic.go`里，约160行代码，相比第一个主要是优化了输出的结果。例如，它将“+ -”替换成“-”，将“1x”替换成“x”，去掉系数为0的项（例如“0x”等），使用`math/cmplx`包里的`cmplx.IsNaN()`函数将一个虚数部分近似0的解转换成浮点数，等等。此外，还用了一些高级的字符串格式化技巧（主要在3.5节介绍）。

[1].这里的实现基于
[http://en.literateprograms.org/Pi_with_Machin's_formula_\(Python\)_](http://en.literateprograms.org/Pi_with_Machin's_formula_(Python)_)

[2].相比之下，在工程上以及Python语言中，虚数用j来表示。

第3章 字符串

本章讲解了Go语言的字符串类型，以及标准库中与字符串类型相关的关键包。本章中各小节的内容包括如何写字面量字符串以及如何使用字符串操作符，如何索引和切片字符串，如何格式化字符串、数值和其他内置类型甚至是自定义类型的输出。

Go语言的高级字符串处理相关的功能几乎每天都要用到，如一个字符一个字符迭代字符串的`for...range`循环，`strings`包和`strconv`包中的函数以及Go语言切片字符串的功能。尽管如此，本章还会深入讲解Go语言的字符串，包括一些底层细节，如字符串类型的内部表示。底层方面的东西非常有趣，并且有时非常有用。

一个Go语言字符串是一个任意字节的常量序列。大部分情况下，一个字符串的字节使用UTF-8编码表示Unicode文本（详见上文中的“Unicode编码”一栏）。Unicode编码的使用意味着Go语言可以包含世界上任意语言的混合，代码页没有任何混乱与限制。

Go语言的字符串类型在本质上就与其他语言的字符串类型不同。Java的String、C++的`std::string`以及Python 3的`str`类型都只是定宽字符序列，而Go语言的字符串是一个用UTF-8编码的变宽字符序列，它的每一个字符都用一个或多个字节表示。

初次接触时可能会觉得这些其他语言的字符串类型比Go语言的字符串类型更加方便，因为它们的字符串中的单个字符可以被字节索引，这在Go语言中只有在字符串只包含7位的ASCII字符（因为它们都用一个单一的UTF-8字节表示）时才可能。但在实际情况下，这从来

都不是个问题。首先，直接索引使用得不多，而Go语言支持一个字符一个字符的迭代；其次，标准库提供了大量的字符串搜索和操作函数；最后，我们随时都可以将Go语言的字符串转换成一个Unicode码点切片（其类型为[]rune），而这个切片是可以直接索引的。

虽然Java或者Python两者也都有提供Unicode编码的字符串，但与这些语言的字符串类型相比，Go语言使用UTF-8编码有更多的优点。Java使用码点序列来表示字符串，每一个字符串占用16位；2.x版本到3.2版本的Python使用类似的方法，只是不同的方式编译的Python使用的是16位或者32位字符。对于英文文本，这意味着Go语言使用8位来表示每一个字符，Java或者Python则至少两倍于此。UTF-8编码的另一个优点是，无需关心机器码的排列顺序，而UTF-16和UTF-32编码的字符串需要知道机器码的排列顺序以便将文本正确地解码。其次，由于UTF-8是世界上文本文件的编码标准，其他语言必须通过编码解码该文件的方式来从其内部编码格式转换过来，而Go语言能够直接读或者写这些文件。此外，有些主要的库（如GTK+）也原生使用UTF-8编码的字符串，因此Go语言无需编码解码就可以使用它们。

Unicode编码

在Unicode编码出现之前，要在单个文件中包含多种语言的文本几乎是不可能的，比如在英文中引用某些日文或者俄文。因为每种语言使用的编码方式不一样，而一个文本文件只支持一种编码方式。

Unicode被设计成能够表示世界上各种写作系统的字符，因此一个使用Unicode编码的单一文件可以包含任意种语言的混合体，包括数学符号、“修饰符”以及其他特殊字符。

每一个Unicode字符都有一个唯一的叫做“码点”的标识数字。目前定义了超过10万个Unicode字符，其码点的值从0x0到0x10FFFF（后者在Go语言中被定义成一个常量`unicode.MaxRune`），其中有一些断层

和许多特殊的情况。在Unicode文档中，码点是用4个或者更多个十六进制数字以U+hhhh的形式表示的，如U+21D4表示↔字符。

在 Go 语言中，一个单一的码点在内存中以 `rune` 的形式表示。（`rune` 类型是 `int32` 类型的别名，详见2.3.1节。）

无论是在文件里还是内存里，Unicode 文本都必须用统一的编码方式表示。Unicode 标准定义了一些Unicode变体格式（编码），如UTF-8、UTF-16以及UTF-32编码。Go语言的字符串类型使用UTF-8编码。UTF-8编码是用得最广的编码，也是文本文件的标准以及XML文件和JSON文件的默认编码方式。

UTF-8编码使用1~4个字节来表示每一个码点。对于只包含7位的ASCII字符的字符串来说，字节和字符之间有一个一对一的关系，因为7位的ASCII字符正好可以用一个UTF-8字节来表示。这样表示的结果是，UTF-8存储英文文本时会非常紧凑（一个字节表示一个字符）的，另一个结果是一个用7位ASCII编码的文本与一个用UTF-8编码的文本没有区别。

在实际使用中，只要我们学会了Go语言中使用字符串的范式，会发现Go语言的字符串与其他语言中的字符串类型一样方便。

[3.1 字面量、操作符和转义](#)

字符串字面量使用双引号（"）或者反引号（'）来创建。双引号用来创建可解析的字符串字面量，如表 3-1 中所示的那些支持转义的序列，但不能用来引用多行。反引号用来创建原生的字符串字面量，这些字符串可能由多行组成；它们不支持任何转义序列，并且可以包含除了反引号之外的任何字符。可解析的字符串使用得最广泛，而原

生的字符串字面量则用于书写多行消息、HTML以及正则表达式。这里有些例子。

```
text1 := " \" what's that?\" , he said " // 可解析的字符串字面量
text2 := ' " what's that? " , he said' // 原生的字符串字面量
radicals := " √\u221A \U0000221a " // radicals == " √√√ "
```

表3-1 Go语言的字符串和字符转义

转义字符	含义
\\	反斜线
\ooo	3 个 8 位数给定的八进制代码的 Unicode 字符
\'	单引号，只用于字符字面量内
\"	双引号，只用于可解析的字符串字面量内
\a	ASCII 码的响铃符
\b	ASCII 码的退格符
\f	ASCII 码的换页符
\n	ASCII 码的换行符
\r	ASCII 码的回车符
\t	ASCII 码的制表符
\uhhhh	4 个 16 位数字给定的十六进制码点的 Unicode 字符
\Uhhhhhhhh	8 个 32 位数字给定的十六进制码点的 Unicode 字符
\v	ASCII 码的垂直制表符
\xhh	2 个 8 位数字给定的十六进制码点的 Unicode 字符

上文中创建的3个变量都是字符串类型，变量text1和变量text2包含的是完全相同的文本。由于.go文件使用的是UTF-8编码，因此我们可以包含Unicode编码字符而无需拘泥于形式。然而我们仍然可以使用Unicode的转义字符来表示第二个或者第三个√字符。但在这个特殊的例子中，我们不能使用八进制或者十六进制的转义符，因为它们的码点仅限于U+0000到U+00FF之间，对于√这个字符的码点U+221A来说太小了。

如果我们想要创建一个长的可解析字符串字面量，但又不想在代码中写同样长的一行，那么我们可以创建多个字面量片段，使用+级联

符将这些片段连接起来。此外，虽然Go语言的字符串是不可变的，但它们支持+=追加操作符。如果底层的字符串容量不够大，不能适应添加的字符串，级联追加操作将导致底层的字符串被替换。这些操作符详见表3-2。字符串也可以使用比较操作符（见表2-3）来进行比较。这里有个例子使用到了这些操作符：

```
book := " The Spirit Level " + // 字符
串级联
    " by Richard Wilkinson "
book += " and Kate Pickett " // 字符
串追加
fmt.Println( " Josey " < " José " , " Josey " == " José " ) // 字符
串比较
```

其结果是book变量将包含文本 " The Spirit Level by Richard Wilkinson and Kate Pickett "，并会输出 " true false " 到os.Stdout。

表3-2 字符串操作符

所有包含7位ASCII字符的字符串都可以使用[]切片操作符。但是如果使用非ASCII操作符则需小心(参见3.4节)。字符串可以使用这些标准的比较操作符<、<=、==、!=、>=、>进行比较(详见表2-3以及3.2节)。

语法	描述/结果
<code>s += t</code>	将字符串 <code>t</code> 追加到字符串 <code>s</code> 末尾
<code>s + t</code>	将字符串 <code>s</code> 和 <code>t</code> 级联
<code>s[n]</code>	字符串 <code>s</code> 中索引位置为 <code>n</code> (uint8 类型) 处的原始字节
<code>s[n:m]</code>	从位置 <code>n</code> 到位置 <code>m-1</code> 处取得的字符串
<code>s[n:]</code>	从位置 <code>n</code> 到位置 <code>len(s)-1</code> 处取得的字符串
<code>s[:m]</code>	从索引位置 0 到位置 <code>m-1</code> 处取得的字符串
<code>len(s)</code>	字符串 <code>s</code> 中的字节数
<code>len([]rune(s))</code>	字符串 <code>s</code> 中字符的个数——可以使用更快的 <code>utf8.RuneCountInString()</code> 来代替, 详见表 3-10
<code>[]rune(s)</code>	将字符串 <code>s</code> 转换成一个 Unicode 码点
<code>string(chars)</code>	将一个 <code>[]rune</code> 或者 <code>[]int32</code> 转换成字符串, 这里假设 <code>rune</code> 和 <code>int32</code> 切片都是 Unicode 码点*
<code>[]byte(s)</code>	无副本地将字符串 <code>s</code> 转换成一个原始字节的切片数组, 不保证转换的字节是合法的 UTF-8 编码字节
<code>string(bytes)</code>	无副本地将 <code>[]byte</code> 或者 <code>[]uint8</code> 转换成一个字符串类型, 不保证转换的字节是合法的 UTF-8 编码字节
<code>string(i)</code>	将任意数字类型的 <code>i</code> 转换成字符串, 假设 <code>i</code> 是一个 Unicode 码点。例如, 如果 <code>i</code> 是 65, 那么其返回值为 "A"*
<code>strconv.Itoa(i)</code>	int 类型 <code>i</code> 的字符串表示和一个错误值。例如, 如果 <code>i</code> 的值是 65, 那么该返回值为 ("65", nil)。详见表 3-8 和表 3-9
<code>fmt.Sprint(x)</code>	任意类型 <code>x</code> 的字符串表示, 例如, 如果 <code>x</code> 是一个值为 65 的数字类型, 那么其返回值为 "65"。详见表 3-3

注: *这种转换总是成功的。非法数字被转换成Unicode编码的替换符U+FFFD, 看起来像“?”。

3.2 比较字符串

如前所述, Go语言字符串支持常规的比较操作 (<、<=、==、!=、>和>=), 这些操作符在表2-3中已给出。这些比较操作符在内存中一个字节一个字节地比较字符串。比较操作可以直接使用,

如比较两个字符串的相等性，也可以间接使用，例如在排序[]string时使用 < 操作符来比较字符串。遗憾的是，执行比较操作时可能会产生3个问题。这3个问题困扰每种使用Unicode字符串的编程语言，都不局限于Go语言。

第一个问题是，有些Unicode编码的字符可以用两个或者多个不同的字节序列来表示。例如，字符Å可以是Ångström中的字符，也可以只是一个A上面加了一个小环，这两者通常不能区分。Ångström字符的Unicode编码是U+212B，但是一个A上面加了一个小圈的字符使用Unicode编码U+00C5来表示，或者使用两个编码U+0041（A）以及U+030A（°，将小圈放到上面）来表示。Ångström中的Å在UTF-8中表示成字节[0xE2, 0X84, 0XAB]，字符Å则表示成字节[0XC3, 0X85]，而一个带有°的A字符则表示成[0X41, 0XCC, 0X81]。当然，从用户的角度看，字符Å应该在比较和排序时都是相等的，无论其底层字节如何表示。

第一个问题并不是我们想象的那样严重，因为所有Go语言中的UTF-8字节序列（即字符串）使用的都是同样的码点到字节的映射。这也意味着，Go语言中的é字符在字符或者字符串字面量中使用同样的字节进行表示。同时，如果我们只关心ASCII字符（即英语），这个问题也就不存在。即便是要处理非ASCII字符，这个问题也仅仅在以下情况下才存在：当我们有两个看起来一样的字符时，或者当我们从一个外部来源中读取UTF-8字节时，这个来源的码点到字节的映射是合法的UTF-8但又不同于Go语言的映射。如果这真的是一个问题，那么也可以写一个自定义的标准化函数来保证不出错。例如，写一个函数使得é总是使用字节[0xC3, 0xA9]（Go语言原生支持这种表示）来表示，而非字节[0x65, 0xCC, 0x81]（即是一个e和一个´组合起来的字符）。Unicode 标准格式文档（unicode.org/reports/tr15）中对如何标准化

Unicode编码字符有详细解释。撰写本文时，Go语言的标准库有一个实验性的标准化包（`exp/norm`）。

由于第一个问题只有当字符串来自于外部源时才可能引起，并且只有当它们使用不同于Go语言的码点到字节的映射时才发生，这个可以通过隔离接收外部字符串的代码来解决。隔离的代码可以在将接收到的字符串提供给程序之前将其标准化。

第二个问题是，有些情况下用户可能会希望把不同的字符看成相同的。例如，我们可能写一个程序来为用户提供文本搜索功能，而用户可能输入单词“file”。通常，用户可能希望搜索所有包含“file”的地方，但用户也可能希望输入所有与“file”（即一个紧跟着“le”的“fi”字符）匹配的地方。类似地，用户可能希望搜索“5”的时候能够匹配“5”、“5”、“5”，甚至是“o5”。与第一个问题一样，这也可以使用一些标准化形式来解决。

第三个问题是，有些字符的排序是与语言相关的。其中一个例子是，瑞典语中的ä在排序时排z之后，但在德国的电话本中排序时拼成ae，而在德国的字典上则被拼成a。另一个例子是，虽然在英文中我们在排序时将其排成o，但在丹麦语和挪威语中，它往往排在z之后。这方面有许许多多的规则，并且由于有时应用程序被不同国家的人使用（因此期望不同的排序规则），有时字符串中混杂着各种语言（如一些西班牙语和英语），有些字符（如箭头、修饰符以及数学符号）基本上就没有实际的排序索引意义，这些规则可能很复杂。

从有利的方面讲，Go语言对字符串按字节比较的方式相当于英文的ASCII排序方式。并且，如果将要比较的字符串转成全部小写或者全部大写，我们可以得到一个更加自然的英语语言顺序，我们将在后面的例子中看到（参见4.2.4节）。

3.3 字符和字符串

在 Go 语言中，字符使用两种不同的方式（可以很容易地相互转换）来表示。一个单一的字符可以用一个单一的 `rune`（或者 `int32`）来表示。从现在开始，我们交替使用术语“字符”、“码点”、“Unicode 字符”以及“Unicode 码点”来表示保存一个单一字符的 `rune`（或者 `int32`）。Go 语言的字符串表示一个包含 0 个或者多个字符序列的串。在一个字符串内部，每个字符都表示成一个或者多个 UTF-8 编码的字节。

我们可以使用 Go 语言的标准转换语法（`string(char)`）将一个字符转换成一个只包含单个字符的字符串。这里有一个例子。

```
æs := " "  
for _, char := range []rune{'æ', 0xE6, 0x346, 0x230, '\xE6', '\u00E6'} {  
    fmt.Printf( " [0x%X '%c'] ", char, char)  
    æs += string(char)  
}
```

这段程序会输出一个行，其中包含 6 个重复的“[0XE6 'æ]”文本。最后，字符串 `æs` 会包含文本 `ææææææ`。（马上我们会看到使用字符串的 `+=` 操作符通过循环来写成的一个更高效的解决方案。）

一个字符串可以使用语法 `chars := []rune(s)` 转换成一个 `rune`（即码点）切片，其中 `S` 是一个字符串类型的值。变量 `chars` 的类型为 `[]int32`，因为 `rune` 是 `int32` 的同义词。这在我们需要逐个字符解析字符串，同时需要在解析过程中能查看前一个或后一个字符时会有用。相反的反转换也同样简单，其语法为 `S:=string(chars)`，其中 `chars` 的类型为 `[]rune` 或者 `[]int32`，得到的 `S` 的类型为字符串。这两个转换都不是无代价的，但这两个转换理论上都比较快（时间代价为 $O(n)$ ，其中 `n` 是字节数，看下文

中的“大O详解”)。更多关于字符串转换的示例请看表3-2。关于数字到字符串的转换情况见表3-8和表3-9。

虽然方便,但是使用+= 操作符并不是在一个循环中往字符串末尾追加字符串最有效的方式。一个更好的方式(Python程序员可能非常熟悉)是准备好一个字符串切片([]string),然后使用strings.Join()函数一次性将其中所有字符串串联起来。但在Go语言中还有一个更好的方法,其原理类似于Java中的StringBuilder。这里有个例子。

```
var buffer bytes.Buffer
for {
    if piece, ok := getNextValidString(); ok {
        buffer.WriteString(piece)
    } else {
        break
    }
}
fmt.Print(buffer.String(), "\n ")
```

我们开始时创建了一个空的bytes.Buffer 类型值。然后使用bytes.Buffer.WriteString()方法将我们需要串联起来的字符串写入到buffer 中(当然,我们也可以在每个字符串之间写入一个分隔符)。最后,bytes.Buffer.String()方法可以用于取回整个级联的字符串(后面我们会看到bytes.Buffer类型的强大功能)。

将一个bytes.Buffer类型中的字符串累加起来可能比+= 操作符在节省内存和操作符方面高效得多,特别是当需要级联的字符串数量很大时。

Go语言的for...range循环(参见5.3节)可以用于一个字符一个字符的迭代字符串,每次迭代都产生一个索引位置和一个码点。下面是一个例子,旁边为其输出。

```

phrase := "vått og tørt"
fmt.Printf("string: \"%s\"\n", phrase)
fmt.Println("index rune char bytes")
for index, char := range phrase {
    fmt.Printf("%-2d    %U  '%c'  %X\n",
        index, char, char,
        []byte(string(char)))
}

```

```

string: "vå tt og tørt"
index rune  char bytes
0      U+0076 'v'   76
1      U+00E5 'å'   C3 A5
3      U+0074 't'   74
4      U+0074 't'   74
5      U+0020 ' '   20
6      U+006F 'o'   6F
7      U+0067 'g'   67
8      U+0020 ' '   20
9      U+0074 't'   74
10     U+00F8 'ø'   C3 B8
12     U+0072 'r'   72
13     U+0074 't'   74

```

大O表示法

大O表示法 $O(\dots)$ 在复杂性理论中是为特定算法所需的处理器和内存消耗给出一个近似边界。大多数都是以 n 的比例来衡量，其中 n 为需要处理的项的数量，或者该项的长度。它们可以用来衡量内存消耗或者处理器的时间消耗。

$O(1)$ 意味着常量时间，也就是说，无论 n 的大小为何，这都是最快的可能。 $O(\log n)$ 意味着对数时间，速度很快，与 $\log n$ 成正比。 $O(n)$ 意味着线性时间，速度也很快，并且与 n 成正比。 $O(n^2)$ （ n 的2次方）意味着二次方时间，速度开始变慢，并且与 n 的平方成正比。 $O(n^m)$ （ n 的 m 次方），意味着多项式时间，随着 n 的增长，它很快就变得很慢，特别是当 $m \geq 3$ 时。 $O(n!)$ 意味着阶乘时间，即使是对于小的 n 值，这在实际使用中也会非常慢。

本书在很多地方都使用大O表示法来方便地解释处理程序的代价，例如，将字符串转换成`[]rune`的代价。

上面程序先创建 `phrase` 字符串字面量，然后在下一行的一个标题之后将其输出。然后我们迭代字符串中的每一个字符。Go语言的`for...range`循环在迭代时将UTF-8字节解码成Unicode码点（`rune`类型），因此我们不必关心其底层实现。对于每一个字符，我们将其索

引位置、码点的值（使用Unicode表示法）、它所表示的字符以及对应的UTF-8字节编码等信息输出。

为了得到一串字节码，我们将码点（`rune` 类型的字符）转换成字符串（它包含一个由一个或者多个 UTF-8 编码字节编码而成的字符）。然后，我们将该单字符的字符串转换成一个`[]byte`切片，以便获取其真实的字节码。其中的`[]byte(string)`转换非常快（ $O(1)$ ），因为在底层`[]byte` 可以简单地引用字符串的底层字节而无需复制。同样，其逆向转换`string([]byte)`的原理也类似，其底层字节也无需复制，因此其代价也是 $O(1)$ 。表3-2列出了Go语言的字符串与字节码之间的相互转换。

我们会马上解释程序中的`%-2d`、`%U`、`%c`以及`%X`格式化声明符（参见3.5节）。如你所见，当 `%X` 声明符用于数字时，它输出该数字的十六进制，当其用于`[]byte` 时，它输出一个含两个十六进制数字的序列，一个数字代表一个字节。这里我们通过/format声明符中加入空格来声明其输出结果需以空格分隔。

在实际的编程中，通过与`strings`包和`fmt`包（以及少数情况下来自于`strconv`、`unicode`、`unicode/utf8`的包）中的函数相配合，使用`for...range`循环来迭代字符串中的字符为处理和操作字符串提供了方便而强大的功能。此外字符串类型还支持切片（因为在底层一个字符串实际上就是一个增强的`[]byte`切片），这非常有用，只要我们小心不将一个多字节的字符切片成一半。

3.4 字符串索引与切片

正如表3-2所示，Go语言支持Python中字符串分割语法的一个子集。我们将在第4章看到，这个语法可以用于任意类型的切片。

由于Go语言的字符串将其文本保存为UTF-8编码的字节，因此我们必须非常小心地只在字符边界处进行切片。这在我们的文本中所包含的字符是7位的ASCII编码字符的情况下非常简单，因为一个字节代表一个字符，但是对于非ASCII文本将更有挑战，因为这些字符可能用一个或者多个字节表示。通常我们完全不需要切片一个字符串，只需使用for...range循环将其一个字符一个字符地迭代，但是有些情况下我们确实需要使用切片来获得一个子字符串。有个能够确定能按字符边界进行切片得到索引位置的方法是，使用Go语言的strings包中的函数如strings.Index()或者strings.LastIndex()。strings包的函数已列在表3-6和表3-7中。

我们将从不同的角度解析字符串。索引位置（即字符串的UTF-8编码字节的位置）从0开始，直到该字符串的长度减1。当然也可以使用从len(s)-n这样的索引形式来从字符串切片的末尾开始往前索引，其中n为从后往前数的字节数。例如，给定一个赋值s := " naïve "，如图3-1给出了其Unicode字符、码点、字节以及一些合法的索引位置和一对切片。

s[:2]		s[2:] == s[len(s)-4:]				切片
'n'	'a'	'i'		'v'	'e'	字符
U+006E	U+0061	U+00EF		U+0076	U+0065	码点
0x6E	0x61	0xC3	0xAF	0x76	0x65	字节
0	1	2	3	4	5	索引
		len(s)-2		len(s)-1		

图3-1 字符串剖析

图3-1所示的每一个位置索引都可以用[]索引操作符来返回其对应的ASCII字符（以字节的形式）。例如，s[0] == 'n'和s[len(s)-1] == 'e'。i字符的起始索引位置为2，但如果我们使用s[2]我们只能够得到编码i字符（0xC3）的第一个UTF-8字节，而这并不是我们想要的。

对于只包含7位ASCII字符的字符串，我们可以使用`s[0]`这样的语法来取得其第一个字符（以字节的形式），也可以使用`s[len(s)-1]`的形式来取得其最后一个字符。然而，通常而言，我们应该使用`utf8.DecodeRuneInString()`来获得第一个字符（作为一个 `rune`，与UTF-8字节数字一起表示该字符），而使用`utf8.DecodeLastRuneInString()`来获得其最后一个字符（详见表3-10）。

如果我们确实需要索引单个字符，也有许多可选的方法。对于只包含7位ASCII字符的字符串，我们只需简单地使用`[]`索引操作符，该查找非常的快速（ $O(1)$ ）。对于包含非 ASCII 字符组成的字符串，我们可以将其转换成`[]rune`再使用`[]`索引操作符。这也提供了非常快速的查找性能（ $O(1)$ ），其代价在于一次性的转换耗费了CPU和内存（ $O(n)$ ）。

在我们的例子中，如果我们这样写`chars := []rune(s)`，那么`chars`变量将被创建为一个包含5个码点的`rune`（即`int32`）切片，而非图3-1中所示的6个字节。同时，我们也讲过可以使用`string(char)`语法很容易地将任何`rune`类型转换成一个包含一个字符的字符串。

对于任意字符串（即那些可能含有非 ASCII 字符的字符串），通过索引来提取其字符通常不是正确的方法。更好的方法是使用字符串切片，它可以很方便地返回一个字符串而非一个字节。为了安全地切片任意字符串，最好使用表3-6和表3-7中介绍的`strings`包中的函数来获得我们需要切片的索引位置。

以下等式对于任意字符串切片都成立，事实上，对于任意类型的切片都成立：

`s == s[:i] + s[i:]` // `s`是一个字符串，`i`是一个整型， $0 \leq i \leq \text{len}(s)$

现在让我们看一个实际的切片例子，其中使用的方法很原始。假设我们有一行文本，并且想从该文本中提取该行的第一个和最后一个字。一个简单的方式是这样写代码：

```

line := " røde og gule sløjfer "
i := strings.Index(line, " ") // 获得第一个空格的索引位置
firstWord := line[:i]         // 从第一个字开始时切片直到
第一个空格
j := strings.LastIndex(line, " ") // 获得最后一个空格
lastWord := line[j+1:]         // 从最后一个空格开始切片到
最后一个字

```

```

fmt.Println(firstWord, lastWord) // 输出: røde sløjfer

```

字符串类型的变量`firstWord`被赋值为字符串`line`中的从索引位置0（第一个字节）开始到索引位置`i-1`（第一个空格之前的字节）之间的字符串，因为字符串切片返回从开始到其结束位置处的字符串，但不包含该结束位置。类似地，`lastWord`被赋值为字符串`line`中从索引位置`j+1`（最后一个空格后面的字节）到`line`结尾处（即到索引位置为`len(line)-1`处）的字符串。

虽然这个实例可以用于处理空格以及所有7位的ASCII字符，但是却不适于处理任意的Unicode空白字符如U+2028（行分隔符）或者U+2029（段落分隔符）。

下面这个例子在以任意空白符分隔字的情况下都可以找出其第一个字和最后一个字。

```

line := " år  tørt\u2028vær "
i := strings.IndexFunc(line, unicode.IsSpace) // i == 3
firstWord := line[:i]
j := strings.LastIndexFunc(line, unicode.IsSpace) // j == 9
_, size := utf8.DecodeRuneInString(line[j:]) // size == 3
lastWord := line[j+size:] // j + size
== 12

```



```
// 打印:  rā
```

如图3-2所示，字符串line以字符、码点以及字节的形式给出。该图也给出了其字节索引位置以及上文代码片段中使用到的切片。

<div><div>line[:i]</div><div>line[j:]</div><div>line[j+size:]</div></div>															切片	
'r'	'ä'	' '	't'	'ø'	'r'	't'	𐀀	'v'	'æ'	'r'	字符					
U+0072	U+00E5	U+0020	U+0074	U+00F8	U+0072	U+0074	U+2028	U+0076	U+00E6	U+0072	码点					
0x72	0xC3 0xA5	0x20	0x74	0xC3 0xB8	0x72	0x74	0xE2 0x80 0xA8	0x76	0xC3 0xA6	0x72	字节					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	索引

图3-2 带空白符的字符串剖析

`strings.IndexFunc()`函数返回作为第一个参数传入的字符串中对于作为第二个参数传入的函数（其签名为 `func(rune)bool`）返回 `true` 时的字符索引位置。函数 `strings.LastIndexFunc()`与此类似，只不过它适于从字符串的结尾处开始工作并返回当函数返回`true`时的最后一个字符索引位置。这里我们传入`unicode`包的`IsSpace()`函数作为其第二个参数，该函数接受一个Unicode码点（其类型为`rune`）作为其唯一的参数，如果该码点是一个空白符则返回`true`（详见表3-11）。一个函数的名字是该函数的引用，因此可以用于传递给另一个需要函数参数的地方，只要该命名函数（即所引用的函数）的签名与声明的参数相符合（参见4.1节）。

使用 `strings.IndexFunc()` 函数来找到第一个空白符，并从头开始到该空白符索引位置的前一位将字符串切片，就可以很容易地得到字符串的第一个字。但是在搜索最后一个空白符的时候就得小心点，因为

有些空白符被编码成不止一个 UTF-8 字节。我们可以通过使用 `utf8.DecodeRuneInString()` 函数解决这个问题，这个函数可以告诉我们字符串切片中起始位置与最后一个空格符的起始位置对应的那个字符所占字节数为多少。然后，我们将这个数字与最后一个空白符所在的索引位置相加，就能够跳过最后一个空白字符，无论用于表示空白字符的字节数为多少，这样我们就能够将最后一个字切片出来。

3.5 使用fmt包来格式化字符串

Go语言标准库中的fmt包提供了打印函数将数据以字符串形式输出到控制台、文件、其他满足`io.Writer`接口的值以及其他字符串中。这些函数已在表 3-3 中列出。有些输出函数返回值为`error`。当将数据打印到控制台时，常常将该错误值忽略，但是如果打印到文件和网络连接等地方时，则一定要检查该错误值 [1] 。

表3-3 fmt包中的打印函数

语法	含义/结果
<code>fmt.Errorf(format, args...)</code>	返回一个包含所给定的格式化字符串以及 <code>args</code> 参数的错误值
<code>fmt.Fprint(writer, args...)</code>	按照格式 <code>%v</code> 和空格分隔的非字符串将 <code>args</code> 写入 <code>writer</code> 中, 返回写入的字节数和一个值为 <code>error</code> 或者 <code>nil</code> 的错误值
<code>fmt.Fprintf(writer, format, args...)</code>	按照字符串格式 <code>format</code> 将 <code>args</code> 参数写入 <code>writer</code> , 返回写入的字节数和一个值为 <code>error</code> 或者 <code>nil</code> 的错误值
<code>fmt.Fprintln(writer, args...)</code>	按照格式 <code>%v</code> 以空格分隔以换行符结尾将参数 <code>args</code> 写入 <code>writer</code> , 返回写入的字节数和一个值为 <code>error</code> 或者 <code>nil</code> 的错误值
<code>fmt.Print(args...)</code>	使用格式 <code>%v</code> 以空格分隔的非字符串将 <code>args</code> 写入 <code>os.Stdout</code> , 返回写入的字节数和一个值为 <code>error</code> 或者 <code>nil</code> 的错误值
<code>fmt.Printf(format, args...)</code>	使用格式化字符串 <code>format</code> 将 <code>args</code> 写入 <code>os.Stdout</code> , 返回写入的字节数和一个值为 <code>error</code> 或者 <code>nil</code> 的错误值
<code>fmt.Println(args...)</code>	使用格式 <code>%v</code> 以空格分隔以换行符结尾将参数 <code>args</code> 写入 <code>os.Stdout</code> , 返回写入的字节数和一个值为 <code>error</code> 或者 <code>nil</code> 的错误值
<code>fmt.Sprint(args...)</code>	返回 <code>args</code> 参数组成的字符串, 每个参数都使用 <code>%v</code> 进行格式化的使用空格分离的非字符串
<code>fmt.Sprintf(format, args...)</code>	返回使用格式 <code>format</code> 格式化的 <code>args</code> 字符串
<code>fmt.Sprintln(args...)</code>	返回使用格式 <code>%v</code> 格式化 <code>args</code> 后的字符串, 以空格分隔以换行符结尾

`fmt`包也提供了一系列扫描函数（如`fmt.Scan()`、`fmt.Scanf()`以及`fmt.Scanln()`函数）用于从控制台、文件以及其他字符串类型中读取数据。其中有些函数将在第8章用到（参见8.1.3.2节）以及表8-2。扫描函数的一种替代是使用`strings.Fields()`函数将字符串分隔为若干字段然后使用`strconv`包中的转换函数将那些非字符串的字段转换成相应的值（如数值），详见表3-8和表3-9。第1章中我们提到，我们可以创建一个`bufio.Reader`通过从`os.Stdin`读取数据来获得用户的输入，然后使用`bufio.Reader.ReadString()`函数来读取用户输入的每一行（参见1.7节）。

输出值的最简单方式是使用 `fmt.Print()` 函数和 `fmt.Println()` 函数（输出到 `os.Stdout`，即控制台），或者使用 `fmt.Fprint()` 函数和 `fmt.Fprintf()` 函数来输出到给定的 `io.Writer`（如一个文件），或者使用 `fmt.Sprint()` 函数和 `fmt.Sprintln()` 函数来输出到一个字符串。

```
type polar struct {radius, 0 float64}
p := polar{8.32,.49}
fmt.Print(-18.5, 17, " Elephant ", -8+.7i, 0x3C7, "\u03C7", " a ", " b
", p)
fmt.Println()
fmt.Println(-18.5, 17, " Elephant ", -8+.7i, 0x3C7, "\u03C7", " a ", "
b ", p)
-18.5·17Elephant(-8+0.7i)·967·967ab{8.32·0.49}
-18.5·17·Elephant·(-8+0.7i)·967·967·a·b·{8.32·0.49}
```

为了清晰起见，特别是当连续输出空格的时候，我们必须在每一个显示的空格之间放入一个字符（`·`）。

`fmt.Print()` 函数和 `fmt.Fprint()` 函数处理空白符的方式与 `fmt.Println()` 函数和 `fmt.Fprintln()` 函数处理空白符的方式略有不同。作为一个经验法则，前者更多地用于输出单个值，或者用于不检查错误值的情况下将某个值转换成字符串（使用 `strconv` 包来做更好的转换），因为它们只在非字符串的值之间输出空格。后者更适用于输出多个值，因为它们会在多个输出值之间加入空格，并在末尾添加一个换行符。

在底层，这些函数都统一使用 `%v` 格式符，并且它们都可以以各种形式打印任何内置的或者自定义的值。例如，这里的打印函数对自定义的 `polar` 类型一无所知，但仍然能够成功地打印 `polar` 的值。

在第6章中，我们将会为自定义类型提供一个 `String()` 方法，这个方法允许我们将该自定义类型以我们期望的方式输出。如果我们想要对

内置类型的打印也拥有类似的控制权，我们可以使用一个将格式化字符串作为第一个参数的打印函数。

用于 `fmt.Errorf()`、`fmt.Printf()`、`fmt.Fprintf()` 以及 `fmt.Sprintf()` 函数的格式字符串包含一个或者多个格式指令，这些格式指令的形式是 `%ML`，其中 `M` 表示一个或者多个可选的格式指令修饰符，而 `L` 则表示一个特定的格式指令字符。这些格式指令已在表 3-4 中列出。有些格式指令可以接收一个或者多个修饰符，这些修饰符已在表 3-5 中列出。

表3-4 `fmt`包中的格式指令

格式指令通常用于输出单个值。如果一个值是一个切片，那么其输出通常是一系列方括号括起来的以空格分隔的值的序列，其中每个值都按格式指令被格式化。如果值是一个映射，可能只需使用 `%v` 或者 `%#v`。但如果键和值都是相同的类型的话，也可以使用与该类型兼容的格式指令。

格式指令	含义/结果
<code>%%</code>	一个 <code>%</code> 字面量
<code>%b</code>	一个二进制整数值（基数为 2），或者是一个（高级的）用科学计数法表示的指数为 2 的浮点数
<code>%c</code>	一个 <code>Unicode</code> 字符的码点值

续表

格式指令	含义/结果
%d	一个十进制数值（基数为 10）
%e	以科学记数法 e 表示的浮点数或者复数值
%E	以科学记数法 E 表示的浮点数或者复数值
%f	以标准记数法表示的浮点数或者复数值
%g	以 %e 或者 %f 表示的浮点数或者复数，任何一个都以最为紧凑的方式输出
%G	以 %E 或者 %f 表示的浮点数或者复数，任何一个都以最为紧凑的方式输出
%o	一个以八进制表示的数字（基数为 8）
%p	以十六进制（基数为 16）表示的一个值的地址，前缀为 0x，字母使用小写的 a~f 表示（用于调试）
%q	使用 Go 语法以及必要时使用转义，以双引号括起来的字符串或者字节切片 []byte，或者是以单引号括起来的数字
%s	以原生的 UTF-8 字节表示的字符串或者 []byte 切片，对于一个给定的文本文件或者在一个能够显示 UTF-8 编码的控制台，它会产生正确的 Unicode 输出
%t	以 true 或者 false 输出的布尔值
%T	使用 Go 语法输出的值的类型
%U	一个用 Unicode 表示法表示的整型码点，默认值为 4 个数字字符。例如，fmt.Printf("%U", '𐀀') 输出 U+00B6
%v	使用默认格式输出的内置或者自定义类型的值，或者是使用其类型的 String() 方法输出的自定义值，如果该方法存在的话
%x	以十六进制表示的整型值（基数为十六），或者是以十六进制数字表示的字符串或者 []byte 数组（每个字节用两个数字表示），数字 a~f 使用小写表示
%X	以十六进制表示的整型值（基数为十六），或者是以十六进制数字表示的字符串或者 []byte 数组（每个字节用两个数字表示），数字 A~F 使用大写表示

表3-5 fmt包中的格式指令修饰符

修饰符	含义/结果
空白	如果输出的数字为负数，则在其前面加上一个减号“-”。如果输出的是正数，则在其前面加上一个空格。使用 %x 或者 %X 格式指令输出时，会在结果之间添加一个空格。例如，fmt.Printf("% X", "←") 输出 E2 86 92
#	让格式指令以另外一种格式输出数据： %#o 输出以 0 打头的八进制数据 %#p 输出一个不含 0x 打头的指针 %#q 尽可能以原始字符串的形式输出一个字符串或者 []byte 切片（使用反引号），否则输出以双引号引起来的字符串

续表

修饰符	含义/结果
#	<code>%#v</code> 使用 Go 语法将值自身输出 <code>%#x</code> 输出以 0x 打头的十六进制数据 <code>%#X</code> 输出以 0X 打头的十六进制数据
+	让格式指令在数值前面输出+号或者-号，为字符串输出 ASCII 字符（别的字符会被转义），为结构体输出其字段名字
-	让格式指令将值进行向左对其（默认值为向右对其）
0	让格式指令以数字 0 而非空白进行填充
<i>n.m</i>	对于数字，这个修饰符会使用 <i>n</i> (int 值) 个字符输出浮点数或者复数（为避免截断可以输出更多个），并在小数点后面输出 <i>m</i> (int 值) 个数字。对于字符串， <i>n</i> 声明了其最小宽度，并且如果字符串的字符太少则会以空格填充，而 <i>m</i> 则声明了输出的字符串所能使用的最长字符个数（从左至右），如果太长则可能会导致字符串被截断。 <i>m</i> 和 <i>n</i> 两个都可以使用 '*' 来代替，这种情况下它们的值就可以从参数中获取。 <i>n</i> 或者 <i>m</i> 都可以被省略

现在让我们来看一些格式化字符串的代表性例子，以便弄清楚它们是如何工作的。在每一个案例中，我们会给出一小段代码以及该代码的输出 [2]。

3.5.1 格式化布尔值

布尔值使用 `%t`（真值，truth value）格式指令来输出。

```
fmt.Printf( " %t %t\n " , true, false)
```

```
true false
```

如果我们想以数值的形式输出布尔值，那么我们必须做这样的转换：

```
fmt.Printf( " %d %d\n " , IntForBool(true), IntForBool(false))
```

```
1 0
```

这里使用了一个小的自定义函数。

```
func IntForBool(b bool) int {
```

```

    if b{
        return 1
    }
    return 0
}

```

我们也可以使用 `strconv.ParseBool()` 函数来将字符串转换回布尔值。当然，将字符串转换成数字也有类似的函数（参见3.6.2节）。

3.5.2 格式化整数

现在让我们来看看整数的格式化，从二进制数字（基数为2）的输出开始。

```
fmt.Printf( " |%b|%9b|%-9b|%09b|% 9b|\n " , 37, 37, 37, 37, 37)
```

```
|100101|...100101|100101...|000100101|...100101|
```

第一个格式（`%b`）使用`%b`（二进制）格式指令，它使用尽量少的数字将一个整数以二进制的形式输出。第二个格式（`%9b`）声明了一个长度为9的字符（为了防止截断，可能会超出输出时所需要的长度），并且使用了默认的右对齐符。第三个格式（`%-9b`）使用`-`修饰符来左对齐。第四个格式（`%09b`）使用`0`作为填充符，第五个格式（`% 9b`）使用空格作为填充符。

八进制格式类似于二进制，但支持另一种格式。它使用 `%o`（八进制，`octal`）格式指令。

```
fmt.Printf( " |%o|%#o| %# 8o| %#+ 8o| %+08o|\n " , 41, 41, 41, 41, -41)
```

```
|51|051|.....051|.....+051|-0000051|
```


使用 `#` 修饰符可以切换格式，从而在输出的时候以 `0` 打头。`+` 修饰符会强制输出正号，如果没有该修饰符，正整数输出时前面没有正号。

十六进制格式使用 `%x` 和 `%X` 格式指令，选择哪个取决于希望将16进制中的A到F字母以小写还是大写表示。

```
i := 3931
```

```
fmt.Printf( " |%x|%X|%8x|%08x|%#04X|0x%04X|\n " , i, i, i, i, i, i)
```

```
|f5b|F5B|.....f5b|00000f5b|0X0F5B|0x0F5B|
```

对于十六进制数字，变更格式修饰符（`#`）将导致输出时以`0x`或者`0X`开头。对于所有的数字，如果我们声明了一个比所需更宽的宽度，输出时会输出额外的空格以便将数字右对齐。如果所声明的宽度太小，则将整个数字输出，因此没有截断的风险。

十进制的数字使用`%d`（十进制，**decimal**）格式指令。唯一可用于当做填充符的字符是空格和`0`，但也容易使用自定义的函数来填充别的字符。

```
i = 569
```

```
fmt.Printf( " |$%d|$%06d|$%+06d|$%s|\n " , i, i, i, Pad(i, 6, '*'))
```

```
|$569|$000569|$+00569|$***569|
```

在最后一种格式中，我们使用`%s`（字符串，**string**）格式指令来输出一个字符串，因为那就是我们的`Pad()` 函数所返回的。

```
func Pad(number, width int, pad rune) string {  
    s := fmt.Sprint(number)  
    gap := width - utf8.RuneCountInString(s)  
    if gap > 0 {
```

```

        return strings.Repeat(string(pad), gap) + s
    }
    return s
}

```

`utf8.RuneCountInString()`函数返回给定字符串的字符数。这个数字永远小于或等于其字节数。`strings.Repeat()`函数接收一个字符串和一个计数，返回一个将该字符串重复给定次数后产生的字符串。我们选择将填充符以`rune`（即Unicode码点）的方式传递以防止该函数的用户传入包含不止一个字符的字符串。

[3.5.3 格式化字符](#)

Go语言的字符都是`rune`（即`int32`值），它们可以以数字或者Unicode字符的形式输出。

```
fmt.Printf( " %d %#04x %U '%c'\n " , 0x3A6, 934, '\u03A6',
'\U000003A6')
```

```
934·0x03a6·U+03A6·'Φ'
```

这里我们以十进制和十六进制的形式输出了一个大写的希腊字母Phi（“Φ”），使用`%U`格式指令来输出Unicode码点，以及使用`%c`（字符或者码点）格式指令来输出Unicode字符。

[3.5.4 格式化浮点数](#)

浮点数格式可以指定整体长度、小数位数，以及使用标准计数法还是科学计数法。

```
for _, x := range []float64{-0.258, 7194.84, -60897162.0218, 1.500089e-8} {
```

```

    fmt.Printf( " |%20.5e|%20.5f|%s|\n " , x, x, Humanize(x, 20, 5, '*',
';'))
}

```

```

|.....-2.58000e-01|.....-0.25800|*****-0.25800|
|.....7.19484e+03|.....7194.84000|*****7,194.84000|
|.....-6.08972e+07|.....-60897162.02180|***-60,897,162.02180|
|.....1.50009e-08|.....0.00000|*****0.00000|

```

这里我们使用一个for...range循环来迭代一个float64类型切片中的数字。

自定义的函数 Humanize()返回一个该数字的字符串表示，该表示法包含了分组分隔符和填充符。

```

func Humanize(amount float64, width, decimals int, pad, separator
rune) string {
    dollars, cents := math.Modf(amount)
    whole := fmt.Sprintf( " %+.0f " , dollars)[1:] // 去除 " ± "
    fraction := " "
    if decimals > 0 {
        fraction = fmt.Sprintf( " %+.*f " , decimals, cents)[2:] // 去除 "
±0 "
    }
    sep := string(separator)
    for i := len(whole) - 3; i > 0; i -= 3 {
        whole = whole[:i] + sep + whole[i:]
    }
    if amount < 0.0 {
        whole = " - " + whole
    }
}

```

```

number := whole + fraction
gap := width - utf8.RuneCountInString(number)
if gap > 0 {
    return strings.Repeat(string(pad), gap) + number
}
return number
}

```

`math.Modf()`函数将一个`float64`类型的数的整数部分和小数部分以两个`float64`类型的数的形式返回。为了以字符串的形式得到其整数部分，我们使用带正号格式的`fmt.Sprintf()`函数强制输出正号，然后立即将其切片以去除正号。针对小数部分，我们也使用类似的技术，只是这次我们使用`.m`格式指令修饰符来声明需要使用*占位符的小数位数（因此在本例中，如果小数的值为2，那么其有效格式为`%+.2f`）。对于小数部分，我们会去除其头部的`-0`或者`+0`。

组分分隔符从右至左插入整个字符串中，如果数字为负值，则插入一个`-`符号。最后，我们将整个结果串联起来并返回，如果位数不够则填充。

`%e`、`%E`、`%f`、`%g`和`%G`格式指令既可以用于复数，也可以用于浮点数。`%e`和`%E`是科学计算法格式（指数的）格式指令，`%f`是浮点数格式指令，而`%g`和`%G`则是通用的浮点数格式指令。

然而，需要注意的一点是，修饰符会分别作用于复数的实部和虚部。例如，如果参数是一个复数，`%6f`格式产生的结果会占用至少20个字符。

```

for _, x := range []complex128{2 + 3i, 172.6 - 58.3019i, -.827e2 +
9.04831e-3i} {
    fmt.Printf( " |%15s|%9.3f|%.2f|%.1e|\n " ,
        fmt.Sprintf( " %6.2f%+.3fi " , real(x), imag(x)), x, x, x)
}

```

```

}
|...2.00+3.000i|(...2.000...+3.000i)|(2.00+3.00i)|(2.0e+00+3.0e+00i)|
|·172.60-58.302i|(·172.600··-58.302i)|(172.60-58.30i)|(1.7e+02-
5.8e+01i)|
|·-82.70+0.009i|(·-82.700...+0.009i)|(-82.70+0.01i)|(-8.3e+01+9.0e-
03i)|

```

对于第一组复数，我们希望小数点后输出不同数量的数字。为此，我们需要使用 `fmt.Sprintf()` 分别格式化复数的实部和虚部部分，然后以 `%15s` 格式将结果以字符串的形式输出。对于其他组的复数，我们直接使用 `%f` 和 `%e` 格式指令，它们总会在输出的复数两边加上圆括号。

3.5.5 格式化字符串和切片

字符串输出时可以指定一个最小宽度（如果字符串太短，打印函数会以空格填充）或者一个最大输出字符数（会将太长的字符串截断）。字符串可以以 Unicode 编码（即字符）、一个码点序列（即 rune）或者表示它们的 UTF-8 字节码的形式输出。

```
slogan := " End Ó ré ttlæti♥ "
```

```
fmt.Printf( " %s\n%q\n%+q\n%#q\n " , slogan, slogan, slogan, slogan)
```

```
End Ó ré ttlæti♥
```

```
" End Ó ré ttlæti♥ "
```

```
" End \u00d3r\u00e9ttl\u00e6ti\u2665 "
```

```
'End Ó ré ttlæti♥'
```

`%s` 格式指令用于输出字符串，我们将很快提到它。`%q`（引用字符串）格式指令用于以 Go 语言的双引号形式输出字符串，其中会直接将可打印字符的可打印字面量输出，而其他不可打印字符则使用转义的形式输出（见表 3-1）。如果使用了 `+` 号修饰符，那么只有 ASCII 字符

（从U+0020到U+007E）会直接输出，而其他字符则以转义字符形式输出。如果使用了#修饰符，那么只要在可能的情况下就会输出Go原始字符串，否则输出以双引号引用的字符串。

虽然通常与一个格式指令相对应的变量是一个兼容类型的单一值（例如int型值相对应的%d或者%x），该变量也可以是一个切片数组或者一个映射，如果该映射的键与值与该格式指令都是兼容的（比如都是字符串或者数字）。

```
chars := []rune(slogan)
fmt.Printf( " %x\n%#x\n%#X\n ", chars, chars, chars)
[45·6e·64·20·d3·72·e9·74·74·6c·e6·74·69·2665]
[0x45·0x6e·0x64·0x20·0xd3·0x72·0xe9·0x74·0x74·0x6c·0xe6·0x74·0x69·0x2665]
[0X45·0X6E·0X64·0X20·0XD3·0X72·0XE9·0X74·0X74·0X6C·0XE6·0X74·0X69·0X2665]
```

这里我们使用%x和%X格式指令以十六进制数字序列的形式打印了一个rune类型的切片，在本例中是一个码点切片，一个十六进制数字对应一个码点。

对于大多数类型，该类型的切片被输出时都会以方括号包围并以空格分隔。其中有个例外，[]byte只有在使用%v格式指令时才会输出方括号和空格。

```
bytes := []byte(slogan)
fmt.Printf( " %s\n%x\n%X\n% X\n%v\n ", bytes, bytes, bytes, bytes, bytes)
```

```
End·Óréttlæti♥
456e6420c39372c3a974746cc3a67469e299a5
456E6420C39372C3A974746CC3A67469E299A5
45·6E·64·20·C3·93·72·C3·A9·74·74·6C·C3·A6·74·69·E2·99·A5
```

```
[69·110·100·32·195·147·114·195·169·116·116·108·195·166·116·105·226·153·165]
```

一个字节切片（这里是表示字符串的UTF-8 字节）可以以十六进制两位数序列的形式输出，其中一个数字表示一个字节。如果我们使用%s 格式指令，则字节切片会被假设为 UTF-8 编码的Unicode，并且以字符串的形式输出。虽然[]bytes类型没有可选的十六进制格式，但这些数字可以像上面倒数第二行所输出的那样使用空格分隔。格式指令%v 以一个方括号包围并以空格分隔的十进制值的形式输出[]bytes类型的值。

Go语言默认是居右对齐，我们可以使用-修饰符来将其居左对齐。当然，我们可以为像下面的例子所示范的那样，指定一个最小的域宽以及一个最大的字符数。

```
s := " Dare to be naive "
fmt.Printf( " |%22s|%-22s|%10s|\n " , s, s, s)
```

```
|·····Dare·to·be·naï ve|Dare·to·be·naï ve·····|Dare·to·be·naive|
```

在这段代码中，第三个格式（%10s）指定了最小域宽为10个字符，但因为字符串的长度比这个域宽要长（该域宽为最小值），所以字符串被完整打印出来。

```
i := strings.Index(s, " n ")
fmt.Printf( " |%.10s|%.10s|%-22.10s|%s|\n " , s, i, s, s)
```

```
|Dare·to·be|Dare·to·be|Dare·to·be··········|Dare·to·be·naive|
```

这里，第一个格式（%.10s）声明了最多打印字符串的10个字符，因此这里输出的字符串被截断成指定的宽度。第二个格式（%.10s）希望输入两个参数——所打印字符个数的最大值和一个字符串，这里我

们使用了字符串的第n个字符的索引位置来作为最大值，这意味着其索引位置小于该值的字符都将被打印出来。第三个格式（%-22.10s）同时声明了最小域宽度为22 个字符和最大输出字符个数为10字符，这也意味着在一个宽为22字符的域中最多只输出该字符串的前10个字符，由于其域宽比要打印的字符数大，因此该域使用空格填充，同时使用 - 修饰符来将其居左对齐。

3.5.6 为调试格式化

%T（类型）格式指令用于打印一个内置的或者自定义值的类型，而%v格式指令则用于打印一个内置值的值。事实上，%v也可以打印自定义类型的值，对于没有定义String()方法的值使用默认的格式，对于定义了String()方法的值则使用该方法打印。

```
p := polar{-83.40, 71.60}
fmt.Printf( " |%T|%v| %#v\n " , p, p, p)
fmt.Printf( " |%T|%v|%t\n " , false, false, false)
fmt.Printf( " |%T|%v|%d\n " , 7607, 7607, 7607)
fmt.Printf( " |%T|%v|%f\n " , math.E, math.E, math.E)
fmt.Printf( " |%T|%v|%f\n " , 5+7i, 5+7i, 5+7i)
s := " Relativity "
fmt.Printf( " |%T\ " %v\ " \ " %s\ " |%q\n " , s, s, s, s)
|main.polar|{-83.4·71.6}|main.polar{radius:-83.4,·θ:71.6}| |
|bool|false|false|
|int|7607|7607|
|float64|2.718281828459045|2.718282|
|complex128|(5+7i)|(5.000000+7.000000i)|
|string| " Relativity " | " Relativity " | " Relativity " |
```


上面这个例子给出了如何使用%T和%v来输出任意值的类型和值。如果满足%v格式指令的格式，那么我们可以简单地使用fmt.Print()或者类似的使用%v作为默认格式的函数。与%v一起使用可选的格式化格式指令修饰符#只对结构体类型起作用，这使得结构体输出它们的类型名字和字段名字。对于浮点数，%v格式更像%g格式指令而非%f格式指令。%T格式在调试方面非常有用，对于自定义类型可以包含其包名（本例中是main）。对字符串使用%q格式指令可以将它们放入引号中方便调试。

Go语言中有两种类型是同义的：uint8和byte，int32和rune。处理int不能处理的32位有符号整数（例如读写二进制文件）时使用int32，处理Unicode码点时使用rune（字符）。

```
s := " Alias ↔ Synonym "
chars := []rune(s)
bytes := []byte(s)
fmt.Printf( " %T: %v\n%T: %v\n ", chars, chars, bytes, bytes)
[]int32: [65 108 105 97 115 8596 83 121 110 111 110 121 109]
[]uint8: [65 108 105 97 115 226 134 148 83 121 110 111 110 121 109]
```

如上例说明的那样，%T格式指令总是输出其原始类型名，而非其同义词。由于字符串中包含一个非ASCII的字符，因此很明显可以发现我们创建了一个rune切片（码点）和一个UTF-8编码的字节切片。

我们也可以使用%p格式指令来输出任意值的地址。

```
i := 5
f := -48.3124
s := " Tomás Bretón "
fmt.Printf( " |%p → %d| %p → %f| %p → %s\n ", &i, i, &f, f, &s, s)
```

|0xf840000300·→·5|0xf840000308·→·-48.312400|f840001990·→·
Tomás·Bretón|

&地址操作符将在下一章介绍（参见4.1节）。如果我们使用%p格式指令和#修饰符，则会将地址开头处的0x剔除掉。这样输出的地址对于调试非常有帮助。

Go语言的输出切片和映射的功能对调试非常有用，正如输出通道的功能一样，也就是说我们可以输出该通道支持发送和接收的类型以及该通道的内存地址。

```
fmt.Println([]float64{math.E, math.Pi, math.Phi})  
fmt.Printf( " %v\n " , []float64{math.E, math.Pi, math.Phi})  
fmt.Printf( " %#v\n " , []float64{math.E, math.Pi, math.Phi})  
fmt.Printf( " %.5f\n " , []float64{math.E, math.Pi, math.Phi})  
[2.718281828459045·3.141592653589793·1.618033988749895]  
[2.718281828459045·3.141592653589793·1.618033988749895]  
[]float64{2.718281828459045,·3.141592653589793,·1.6180339887498  
95}  
[2.71828·3.14159·1.61803]
```

使用未修饰的%v格式指令，切片可以以方括号包围并将每一项以空格分隔的形式输出。通常我们使用类似fmt.Print()和fmt.Sprint()这样的函数将其输出，但如果我们使用一个格式化的输出函数，那么其常用的格式指令是%v或者%#v。然而，我们也可以使用一个类型兼容的格式指令，如用于浮点数的%f和用于字符串的%s。

```
fmt.Printf( " %q\n " , []string{ " Software patents " , " kill " , " innovation " })
```

```
fmt.Printf( " %v\n " , []string{ " Software patents " , " kill " , " innovation " })
```

```
fmt.Printf( " %#v\n " , []string{ " Software patents " , " kill " , " innovation " })
```

```
fmt.Printf( " %17s\n " , []string{ " Software patents " , " kill " , " innovation " })
```

```
[ " Software·patents " · " kill " · " innovation " ]  
[Software·patents·kill·innovation]  
[]string{ " Software·patents " ,· " kill " ,· " innovation " }  
[·Software·patents······kill······innovation]
```

当字符串中包含空格时，使用%q格式指令来输出字符串切片非常有用，因为这使得每个单个的字符串都是可识别的。使用%v格式指令无法做到这点。

最后一个输出初看起来可能有误，因为它占用了 53 个字符（不包括两边的方括号）而非51个（3个17字符的字符串，都不大）。这个明显的差异在于输出的每一个切片项目之间的空格分隔符。

为了更好地调试，使用%#v格式指令可以以编程的形式输出Go语言代码。

```
fmt.Printf( " %v\n " , map[int]string{1: " A " , 2: " B " , 3: " C " , 4: " D " })
```

```
fmt.Printf( " %#v\n " , map[int]string{1: " A " , 2: " B " , 3: " C " , 4: " D " })
```

```
fmt.Printf( " %v\n " , map[int]int{1: 1, 2: 2, 3: 4, 4: 8})
```

```
fmt.Printf( " %#v\n " , map[int]int{1: 1, 2: 2, 3: 4, 4: 8})
```

```
fmt.Printf( " %04b\n " , map[int]int{1: 1, 2: 2, 3: 4, 4: 8})
```

```
map[4:D·1:A·2:B·3:C]  
map[int]·string{4: " D " ,·1: " A " ,·2: " B " ,·3: " C " }  
map[4:8·1:1·2:2·3:4]  
map[int]·int{4:8,·1:1,·2:2,·3:4}
```

```
map[0100:1000·0001:0001·0010:0010·0011:0100]
```

映射的输出内容以关键字“map”开头，然后是该映射的“键/值”对（以任意的顺序，因为映射是无序的）。正如切片一样，它也可以使用除%v之外的格式指令输出，但只限于其键和值与该格式指令相兼容的情况，正如本例中最后一条语句中那样。（映射和切片将在第4章详细阐述。）

fmt包的输出函数功能非常丰富，并且可以用于输出任意我们想要的东西。该包唯一没有提供的功能是以某种特定的字符串进行填充（而非0或者空格），但正如我们所看到的Pad()（参见3.5.2节）和Humanize()（参见3.5.4节）函数一样，要做到这些也非常简单。

3.6 其他字符处理相关的包

Go语言处理字符串的强大之处不仅限于对索引和切片的支持，也不限于fmt的格式化功能。strings包提供了非常强大的功能，此外strconv、unicode/utf8、unicode等也提供了大量实用的函数，这一节出现的就不少。这本书有好几个地方都用到了regexp提供的正则表达式，本节后面也有介绍。

除此之外，标准库里还有很多其他的包同样提供了字符串相关的功能，其中有一些在我们这本书的例子和习题里经常用到。

3.6.1 strings包

一个常见的字符串处理场景是，我们需要将一个字符串分隔成几个字符串后再做其他处理（例如转换成数字或者过滤空格等）。

为了让大家知道怎么去使用**strings**包里的函数，我们来看一些非常简单的使用示例。表3-6和表3-7里列出了**strings**包里所有的函数。首先，我们从分隔一个字符串开始：

```
names := " Niccolò•Noël•Geoffrey•Amélie••Turlough•José "
fmt.Print( " | " )
for _, name := range strings.Split(names, " • " ) {
    fmt.Printf( " %s| " , name)
}
fmt.Println()
```

```
|Niccolò|Noël|Geoffrey|Amélie||Turlough|José|
```

names 是一个使用圆点符号分隔的名字列表（注意，有一个名字是空的）。我们使用**strings.Split()**函数来切分它，这个函数可以将一个字符串按照指定的分隔符全部切分开，使用**strings.SplitN()**可以指定切的次数（从左到右）。如果使用**strings.SplitAfter()**函数的话输出结果是这样的：

```
|Niccolò•|Noël•|Geoffrey•|Amélie••|Turlough•|José|
```

函数**strings.SplitAfter()** 执行的操作和**strings.Split()**是一样的，但是保留了分隔符。同理， **strings.SplitAfterN()**函数可以指定切割的次数。

如果我们想按两个或更多字符进行切分，可以使用**strings.FieldsFunc()**函数。

```
for _, record := range []string{ " László Lajtha*1892*1963 " ,
    " Édouard Lalo\t1823\t1892 " , " José Ángel Lamas|1775|1814 " }
{
    fmt.Println(strings.FieldsFunc(record, func(char rune) bool {
```

```

switch char {
case '\t', '*', '|':
    return true
}
return false
}))
}

```

[László·Lajtha·1892·1963]

[Édouard·Lalo·1823·1892]

[José·Ángel·Lamas·1775·1814]

`strings.FieldsFunc()` 函数有两个参数，一个字符串（这个例子里是 `record` 变量），一个签名为 `func(rune) bool` 的函数引用。因为这个函数很小而且只用在地方，所以我们直接在调用它的地方创建了一个匿名函数（用这种方式创建的函数称之为闭包，不过在这里我们并没有用到引用环境，参见 5.6.3 节）。`strings.FieldsFunc()` 函数遍历字符串并将每一个字符作为参数传递给函数引用，如果该函数返回 `true` 则执行切分操作。从上面的代码我们可以看出，程序在遇到缩进符号、星号或者竖线的地方进行切分。（Go 语言的 `switch` 语句在 5.2.2 节介绍。）

使用 `strings.Replace()` 函数，我们可以将在一个字符串中出现的某个字符串全部替换成另一个，例如：

```

names = " Antônio\tAndré\tFriedrich\t\tJean\t\tÉlisabeth\tIsabella \t
"

names = strings.Replace(names, "\t", " ", -1)
fmt.Printf( "%s\n", names)

```

|·Antônio·André·Friedrich··Jean·Élisabeth·Isabella·|

`strings.Replace()`的参数有原字符串、被替换的字符串、用来替换的字符串，还有一个指定要替换（从左到右）的次数（-1 表示没有限制），返回一个完成替换的字符串（替换结果不会相互交叠）。

表3-6 `strings`包里的函数列表 #1

变量 `s` 和 `t` 都是字符串类型，`xs` 是字符串切片，`i` 是 `int` 型，`f` 是一个签名为 `func (rune)bool` 的函数引用。索引位置是指位置匹配 Unicode 码点或者字符串的第一个 UTF-8 字节的位置，如果没找到匹配的字符串则为-1。

语法	含义/结果
<code>strings.Contains(s, t)</code>	如果 <code>t</code> 在 <code>s</code> 中则返回 <code>true</code>
<code>strings.Count(s, t)</code>	<code>t</code> 在 <code>s</code> 中出现了多少次
<code>strings.EqualFold(s, t)</code>	如果字符串相等的话则返回 <code>true</code> ，注意此函数比较时是区分大小写的
<code>strings.Fields(s)</code>	在字符串空白处进行切分，返回字符串切片
<code>strings.FieldsFunc(s, f)</code>	按照 <code>f</code> 函数的返回结果进行切分，如果 <code>f</code> 返回 <code>true</code> ，就在那个字符上进行切分
<code>strings.HasPrefix(s, t)</code>	如果字符串 <code>s</code> 是以 <code>t</code> 开头的则返回 <code>true</code>
<code>strings.HasSuffix(s, t)</code>	如果字符串 <code>s</code> 是以 <code>t</code> 结尾的则返回 <code>true</code>
<code>strings.Index(s, t)</code>	<code>t</code> 在 <code>s</code> 中第一次出现的索引位置
<code>strings.IndexAny(s, t)</code>	<code>s</code> 中第一个出现在 <code>t</code> 中的字符的索引位置
<code>strings.IndexFunc(s, f)</code>	<code>s</code> 中第一次令 <code>f</code> 函数返回 <code>true</code> 的字符的索引位置
<code>strings.IndexRune(s, char)</code>	返回字符 <code>char</code> 在 <code>s</code> 中第一次出现的索引位置
<code>strings.Join(xs, t)</code>	将 <code>xs</code> 中的所有字符串按照 <code>t</code> 分隔符进行合并（ <code>t</code> 可能为""）

续表

语法	含义/结果
<code>strings.LastIndex(s, t)</code>	<code>t</code> 在 <code>s</code> 中最后一次出现的位置
<code>strings.LastIndexAny(s, t)</code>	<code>s</code> 中最后一个出现在 <code>t</code> 中的字符的索引位置
<code>strings.LastIndexFunc(s, f)</code>	<code>s</code> 中最后一个 <code>f</code> 返回 <code>true</code> 的字符的索引位置
<code>strings.Map(mf, t)</code>	按照 <code>mf</code> 函数规则（ <code>func (rune) rune</code> ）替换 <code>t</code> 中所有对应的字符
<code>strings.NewReader(s)</code>	创建一个字符串 <code>s</code> 的对象，支持 <code>Read()</code> 、 <code>ReadByte()</code> 和 <code>ReadRune()</code> 方法
<code>strings.NewReplacer(...)</code>	创建一个替换器能够处理多对旧新字符串的替换
<code>strings.Repeat(s, i)</code>	重复 <code>i</code> 次字符串 <code>s</code>

表3-7 strings包里的函数列表#2

变量 `r` 是 `unicode` 类型的, `SpecialCase` 是用来指定 `Unicode` 规则的(高级用法)。

语法	含义/结果
<code>strings.Replace(s, old, new, i)</code>	返回一个新的字符串, 对 <code>s</code> 中旧的非重叠字符串用新的字符串进行替换, 执行 <code>i</code> 次替换操作, 如果 <code>i = -1</code> 则全部替换
<code>strings.Split(s, t)</code>	返回一个新的字符串切片, 在原 <code>s</code> 上所有出现 <code>t</code> 的位置进行切分
<code>strings.SplitAfter(s, t)</code>	同上, 但是保留分隔符
<code>strings.SplitAfterN(s, t, i)</code>	同上, 但是只进行前 <code>i</code> 次分割操作
<code>strings.SplitN(s, t, i)</code>	同 <code>strings.Split()</code> , 但是只执行前 <code>i</code> 次分割操作
<code>strings.Title(s)</code>	返回一个新的字符串, 对原字符串中每一个单词进行标题首字母大写处理
<code>strings.ToLower(s)</code>	返回一个新的字符串, 对原 <code>s</code> 进行字母小写转换
<code>strings.ToLowerSpecial(r, s)</code>	返回一个新的字符串, 按照指定的优先规则对原 <code>s</code> 中的相应的 <code>Unicode</code> 字母进行小写转换
<code>strings.ToTitle(s)</code>	返回一个新的字符串, 对原 <code>s</code> 进行标题格式转换
<code>strings.ToTitleSpecial(r, s)</code>	返回一个新的字符串, 对原 <code>s</code> 按照指定的优先规则 <code>r</code> 进行标题格式转换
<code>strings.ToUpper(s)</code>	返回一个新的字符串, 对原 <code>s</code> 中所有的字母进行大写转换处理
<code>strings.ToUpperSpecial(r, s)</code>	返回一个新的字符串, 按照指定的优先规则对原 <code>s</code> 中的相应的 <code>Unicode</code> 字母进行大写转换
<code>strings.Trim(s, t)</code>	返回一个新的字符串, 从 <code>s</code> 两端过滤掉 <code>t</code>
<code>strings.TrimFunc(s, f)</code>	返回一个新的字符串, 从 <code>s</code> 两端开始过滤掉 <code>f</code> 返回 <code>true</code> 的每一个字符
<code>strings.TrimLeft(s, t)</code>	返回一个新的字符串, 从 <code>s</code> 左边开始过滤掉 <code>t</code>

续表

语法	含义/结果
<code>strings.TrimLeftFunc(s, f)</code>	返回一个新的字符串, 从 <code>s</code> 左边开始过滤掉 <code>f</code> 返回 <code>true</code> 的每一个字符
<code>strings.TrimRight(s, t)</code>	返回一个新的字符串, 从 <code>s</code> 右边开始过滤掉 <code>t</code>
<code>strings.TrimRightFunc(s, f)</code>	返回一个新的字符串, 从 <code>s</code> 右边开始过滤掉 <code>f</code> 返回 <code>true</code> 的每一个字符
<code>strings.TrimSpace(s)</code>	返回一个新的字符串, 从 <code>s</code> 左右两端开始过滤掉空格

通常, 当我们接收到一些用户输入或者是外部输入的数据时, 需要处理一下字符串中出现的空白, 比如说去掉首尾的空白字符, 还有

将中间出现的空白用一个简单的空格符来代替等，可以这么做：

```
fmt.Printf( " |%s|\n " , SimpleSimplifyWhitespace(names))
```

```
|Antônio·André·Friedrich·Jean·Élisabeth·Isabella|
```

函数SimpleSimplifyWhitespace() 实际上只有一行代码。

```
func SimpleSimplifyWhitespace(s string) string {  
    return strings.Join(strings.Fields(strings.TrimSpace(s)), " ")  
}
```

其中，strings.TrimSpace() 返回一个去掉首尾空白的字符串。strings.Fields()在字符串空白上进行分隔，返回一个字符串切片。而函数 strings.Join()则将一个字符串切片重新拼凑成一个字符串，并用指定的分隔符隔开(分隔符可以为空，这里我们用了一个空格)。这3个函数的组合使用，就可以实现规范字符串空白的效果。

当然，我们还可以用bytes.Buffer来实现一种更加高效的空白处理方法。

```
func SimplifyWhitespace(s string) string {  
    var buffer bytes.Buffer  
    skip := true  
    for _, char := range s {  
        if unicode.IsSpace(char) {  
            if !skip {  
                buffer.WriteRune(' ')  
                skip = true  
            }  
        } else {  
            buffer.WriteRune(char)  
            skip = false  
        }  
    }  
    return buffer.String()  
}
```

```

    }
}
s = buffer.String()
if skip && len(s) > 0 {
    s = s[:len(s)-1]
}
return s
}

```

从上面的代码我们可知，函数`SimplifyWhitespace()` 遍历输入字符串的每一个字符，使用`unicode.IsSpace()`函数（见表3-11）跳过字符串开头所有的空白，然后将其他字符累加到 `bytes.Buffer` 里去，对于中间出现的所有空白处都用一个简单的空格符替换，原字符串结尾处的空白也会被去掉（算法允许结尾最多只有一个空格），最后返回需要的字符串。后面还有一种使用正则表达式来处理的版本，更加简单（参见3.6.5节）。

`strings.Map()`函数可以用来替换或者去掉字符串中的字符。它需要两个参数，第一个是签名为`func(rune) rune`的映射函数，第二个是字符串。对字符串中的每一个字符，都会调用映射函数，将映射函数返回的字符替换掉原来的字符，如果映射函数返回负数，则原字符会被删掉。

```

asciiOnly := func(char rune) rune {
    if char > 127 {
        return '?'
    }
    return char
}

fmt.Println(strings.Map(asciiOnly, " JérômeÖsterreich "))

```

J?r?me·?sterreich

在这里我们没有像之前的例子 `strings.FieldsFunc()` 那样直接在调用它的地方创建一个匿名函数，而是将一个匿名函数赋值给一个变量 `asciiOnly`（相当于一个函数的引用）。然后我们将变量 `asciiOnly` 和一个待处理的字符串作为参数来调用 `strings.Map()`。最后打印返回的字符串，把原字符串中所有的非ASCII字符都替换为“?”。当然了，我们也可以在直接调用映射函数的地方创建它，但是如果函数太长或者我们需要在多个地方用到它的话，分离它可以提高代码的复用程度。

要把非ASCII编码的字符删除掉然后输出下面这样的结果也是很容易的：

Jrme·sterreich

实现的方法就是修改映射函数，对于非ASCII编码的字符返回“-1”而不是“?”即可。

我们之前提到过可以用 `for...range` 循环（循环语句在5.3节介绍）以Unicode码点的形式来遍历一个字符串中所有的字符。从实现了 `ReadRune()` 方法的类型中读取数据时可以得到类似的效果，例如 `bufio.Reader` 类型。

```
for {
    char, size, err := reader.ReadRune()
    if err != nil { // 如果读者正在读文件可能发生
        if err == io.EOF { // 没有事故结束
            break
        }
    }
    panic(err) // 出现了一个问题
```

```

    }
    fmt.Printf( "  %U  '%c'  %d:  %  X\n  " , char, char, size,
[]byte(string(char)))
}

```

```
U+0043·'C'·1:·43
```

```
U+0061·'a'·1:·61
```

```
U+0066·'f'·1:·66
```

```
U+00E9·'é'·2:·C3·A9
```

这段代码读取一个字符串，输出每个字符的码点、字符本身和这个字符占用了多少个UTF-8字节，还有用来表示这个字符的字节序列。通常情况下`reader`是对文件进行操作，因此我们可能会假设`reader`变量是通过基于一个`os.Open()`调用返回的`reader`调用`bufio.NewReader()`而创建。我们曾在第一章的`americanise` 示例中见过这种用法（参见 1.6 节）。不过在本例中`reader`被创建用于操作一个字符串：

```
reader := strings.NewReader( " Café " )
```

`strings.NewReader()` 返回的`*strings.Reader`实现了`bufio.Reader`的部分功能，包括 `strings.Reader.Read()`、`strings.Reader.ReadByte()`、`strings.Reader.ReadRune()`、`strings.Reader.UnreadByte()`、`strings.Reader.UnreadRune()`等。这种能够操作具有某个特定接口（例如，这个类型实现了 `ReadRune()`方法）的值而不是某个特定类型的值的能力，是Go语言一个非常强大和灵活的特性，这在第6章会有更详尽的介绍。

[3.6.2 strconv包](#)

`strconv`包提供了许多可以在字符串和其他类型的数据之间进行转换的函数。所有的函数都在表3-8和表3-9里（也可以看一下`fmt`包的打

印和扫描函数，分别在3.5节和8.2节有介绍）。我们先来看一个简单的例子。

一种常见的需求是将真值的字符串表示转换成一个 `bool`。这可以使用 `strconv.ParseBool()`函数来实现。

```
for _, truth := range []string{ " 1 " , " t " , " TRUE " , " false " , " F " , " 0 " , " 5 " } {  
    if b, err := strconv.ParseBool(truth); err != nil {  
        fmt.Printf( " \n{%v} " , err)  
    } else {  
        fmt.Print(b, " " )  
    }  
}  
fmt.Println()  
true·true·true·false·false·false  
{strconv.ParseBool:·parsing· " 5 " :·invalid·syntax}
```

表3-8 `strconv`包函数列表 #1

参数 *bs* 是一个 []byte 切片, *base* 是一个进制单位 (2~36), *bits* 是指其结果必须满足的比位数 (对于 int 型的数据而言, 可以是 8、16、32、64 或者是 0。对于 float64 型的数据而言, 可能是 32 或者 64), 而 *s* 是一个字符串。

语法	含义/结果
<code>strconv.AppendBool(bs, b)</code>	根据布尔变量 <i>b</i> 的值, 在 <i>bs</i> 后追加 "true" 或者 "false" 字符
<code>strconv.AppendFloat(bs, f, fmt, prec, bits)</code>	在 <i>bs</i> 后面追加浮点数 <i>f</i> , 其他参数请参考 <code>strconv.Format.Float()</code> 函数
<code>strconv.AppendInt(bs, i, base)</code>	根据 <i>base</i> 指定的进制在 <i>bs</i> 后追加 int64 数字 <i>i</i>
<code>strconv.AppendQuote(bs, s)</code>	使用 <code>strconv.Quote()</code> 追加 <i>s</i> 到 <i>bs</i> 后面
<code>strconv.AppendQuoteRune(bs, char)</code>	使用 <code>strconv.QuoteRune(char)</code> 追加 <i>char</i> 到 <i>bs</i> 后面
<code>strconv.AppendQuoteRuneToASCII(bs, char)</code>	使用 <code>strconv.QuoteRuneToASCII(char)</code> 追加 <i>char</i> 到 <i>bs</i> 后面
<code>Strconv.AppendouotetoASCII(bs, s)</code>	使用 <code>strconv.QuotetoASCII</code> 追加 <i>s</i> 到 <i>bs</i> 后面
<code>strconv.AppendUInt(bs, u, base)</code>	将 uint64 类型的变量 <i>u</i> 按照指定的进制 <i>base</i> 追加到 <i>bs</i> 后面
<code>strconv.Atoi(s)</code>	返回转换后的 int 类型值和一个 error (出错时 error 不为空), 可参考 <code>strconv.ParseInt()</code>
<code>strconv.CanBackquote(s)</code>	检查 <i>s</i> 是否是一个符合 Go 语言语法的字符串常量, <i>s</i> 中不能出现反引号
<code>strconv.FormatBool(tf)</code>	格式化布尔变量 <i>tf</i> , 返回 "true" 或 "false" 字符串
<code>strconv.FormatFloat(f, fmt, prec, bits)</code>	将浮点数 <i>f</i> 格式化成字符串。 <i>fmt</i> 是格式化动作, 一个字节, 如 'b' 表示 %b, 'e' 表示 %e, 等等 (可参见表 3-4)。如果 <i>fmt</i> 指定为 'e'、'E'、'f' 时, <i>prec</i> 参数表示小数点后面至多保留多少位, 或者当 <i>fmt</i> 指定为 'g' 或者 'G' 时, <i>prec</i> = -1 可以获得能用的最少的数字个数, 同时使用其他方法保留精度损失。 <i>bits</i> 通常是 64
<code>strconv.FormatInt(i, base)</code>	将整数 <i>i</i> 以 <i>base</i> 指定的进制形式转换成字符串
<code>strconv.FormatUInt(u, base)</code>	将整数 <i>u</i> 以 <i>base</i> 指定的进制形式转换成字符串
<code>strconv.IsPrint(c)</code>	判断 <i>c</i> 是否为可打印字符
<code>strconv.Itoa(i)</code>	将十进制数 <i>i</i> 转换成字符串, 可参考 <code>strconv.FormatInt()</code>

表3-9 strconv包函数列表 #2

语法	含义/结果
<code>strconv.ParseBool(s)</code>	如果 <i>s</i> 是"1"、"t"、"T"、"true"、"TRUE"则返回 <code>true</code> 和 <code>nil</code> ，如果 <i>s</i> 是"0"、"f"、"F"、"false"、"False" 或者"FALSE"则返回 <code>false</code> 和 <code>nil</code> ，否则返回 <code>false</code> 和一个 <code>error</code>
<code>strconv.ParseFloat(s, bits)</code>	如果 <i>s</i> 能够转换成浮点数，则返回一个 <code>float64</code> 类型的值和 <code>nil</code> ，否则返回 0 和 <code>error</code> ； <i>bits</i> 应该是 64，但是如果转换成 <code>float32</code> 的话可以设置为 32
<code>strconv.ParseInt(s, base, bits)</code>	如果 <i>s</i> 能够转换成整数，则返回 <code>int64</code> 值和 <code>nil</code> ，否则返回 0 和 <code>error</code> ；如果 <i>base</i> 为 0，则表示要从 <i>s</i> 中判断进制的大小（字符串开头是"0x"或者"0X"表示这是十六进制的，开头只有"0"表示八进制，否则其他的都是十进制），或者在 <i>base</i> 中指定进制的大小（2~36）；如果需要转换成 <code>int</code> 型的话 <i>bits</i> 应该为 0，否则将会转换成带有长度的整形（如 <i>bits</i> 为 16 的话将会转换成 <code>int16</code> ）
<code>strconv.ParseUint(s, base, bits)</code>	同上，唯一不同的只是转换成无符号整数
<code>strconv.Quote(s)</code>	使用 Go 语言双引号字符串语法形式来表示一个字符串，参见表 3-1
<code>strconv.QuoteRune(char)</code>	使用 Go 语言单引号字符串语法来表示一个 <code>rune</code> 类型的 Unicode 码字符 <i>char</i>
<code>strconv.QuoteRuneToASCII(char)</code>	同上，但是对于非 ASCII 码字符进行转义
<code>strconv.QuoteToASCII(s)</code>	同 <code>strconv.Quote()</code> ，但是对非 ASCII 码字符进行转义
<code>strconv.Unquote(s)</code>	对于一个用 Go 语法如单引号、双引号、反引号等表示的字符或字符串，返回引号中的字符串和一个 <code>error</code> 变量
<code>strconv.UnquoteChar(s, b)</code>	一个 <code>rune</code> （第一个字符）、一个 <code>bool</code> （表示第一个字符的 UTF-8 表示需要多个字节）、一个 <code>string</code> （剩下的字符串）以及一个 <code>error</code> ；如果 <i>b</i> 被设置为一个单引号或者双引号，那么引号必须被转义

所有的 `strconv` 转换函数返回一个结果和 `error` 变量，如果转换成功的话 `error` 为 `nil`。

```
x, err := strconv.ParseFloat( "-99.7 " , 64)
```

```
fmt.Printf( " %8T %6v %v\n " , x, x, err)
```

```
y, err := strconv.ParseInt( " 71309 " , 10, 0)
```

```

fmt.Printf( " %8T %6v %v\n " , y, y, err)
z, err := strconv.Atoi( " 71309 " )
fmt.Printf( " %8T %6v %v\n " , z, z, err)
·float64·-99.7·<nil>
··int64·71309·<nil>
····int·71309·<nil>

```

上述代码中的`strconv.ParseFloat()`、`strconv.ParseInt()`、`strconv.Atoi()`（ASCII 转换成 `int`）这 3 个函数可以做的事情比我们想象的多。`strconv.Atoi(s)`和`strconv.ParseInt(s, 10, 0)`的作用是一样的，就是将字符串形式表示的十进制数转换成一个整形值，唯一不同的是`Atoi()`返回`int`型而`ParseInt()`返回`int64`类型。顾名思义，`strconv.ParseUint()`函数可以将一个无符号整数转换成字符串，字符串不能以负号开头，否则会转换失败。还要注意的，当字符串开始处或者结尾处包含空白的话，所有的这些函数都会返回失败，但是我们可以使用 `strings.TrimSpace()` 函数来避免这种情况，或者使用`fmt`包里的扫描函数（表8-2中）。此外，浮点数转换还能处理包含数学标记或者指数符号的字符串，例如 " 984 " 、 " 424.019 " 、 " 3.916e-12 " 等。

```

s := strconv.FormatBool(z > 100)
fmt.Println(s)
i, err := strconv.ParseInt( " 0xDEED " , 0, 32)
fmt.Println(i, err)
j, err := strconv.ParseInt( " 0707 " , 0, 32)
fmt.Println(j, err)
k, err := strconv.ParseInt( " 10111010001 " , 2, 32)
true
57069·<nil>
455·<nil>

```


`strconv.FormatBool()`函数根据给定的布尔变量`true`或者`false`返回一个表示布尔表达式的字符串。`strconv.ParseInt()`函数将一个字符串表示的整数转换成`int64`值。第二个参数是用来指定进制大小的，为0的话表示根据字符串前缀来判断，如“0x”、“0X”表示十六进制，“0”表示八进制，其他都是十进制。在上面的例子里，我们根据字符串的前缀自动判断和转换了一个十六进制和一个八进制数，并以明确指定进制为2的方式转换了一个二进制数。进制大小在2到36之间，如果进制大于10则用A或a来表示10，其他以此类推。函数第三个参数是位大小（为0则默认是`int`大小），所以虽然函数总是返回`int64`，但是只有在真正能够转换成指定大小的整数时才会返回成功。

```
i := 16769023
fmt.Println(strconv.Itoa(i))
fmt.Println(strconv.FormatInt(int64(i), 10))
fmt.Println(strconv.FormatInt(int64(i), 2))
fmt.Println(strconv.FormatInt(int64(i), 16))
```

```
16769023
16769023
111111111101111111111111
ffdf
```

函数`strconv.Itoa()`（函数名是“Integer to ASCII”的缩写）将`int`型的整数转换成以十进制表示的字符串。而函数`strconv.FormatInt()`则可以将其转换成任意进制形式的字符串（进制参数一定要指定，必须在2~36这个范围内）。

```
s = " Alle ønsker å være fri. "
quoted := strconv.Quote(s)
fmt.Println(quoted)
```

```
fmt.Println(strconv.Unquote(quoted))
```

```
" Alle·\u00f8nsker·\u00e5·v\u00e6re·fri. "
```

```
Alle·ønsker·å·være·fri.·<nil>
```

函数 `strconv.Quote()` 返回一个字面量字符串，首尾增加了双引号，并对所有不可打印的ASCII字符和非ASCII字符进行转义（Go语言的转义参见表3-1）。`strconv.Unquote()`函数接受的参数为一个双引号字符串或者使用反引号的原生字符串，或者单引号括起来的字符，返回去除引号后的字符串和一个error变量（成功则为nil）。

3.6.3 utf8包

`unicode/utf8` 有几个很有用的函数，主要用来查询和操作UTF-8 编码的字符串或者字节切片，参见表 3-10。之前我们已经知道如何使用 `utf8.DecodeRuneString()` 函数和 `utf8.DecodeLastRuneInString()` 函数来获得一个字符串的首尾字符。

表3-10 utf8包

使用 `utf8` 包里的函数需要在程序中导入“`unicode/utf`”，变量 `b` 是一个 `[]byte` 类型的切片，`s` 是字符串，`c` 是一个 `rune` 类型的 Unicode 码点。

语法	含义/结果
<code>utf8.DecodeLastRune(b)</code>	返回 <code>b</code> 中最后一个 <code>rune</code> 和它占用的字节数，或者 <code>U+FFFD</code> （Unicode 替换字符？）和 0，如果 <code>b</code> 最后一个 <code>rune</code> 是非法的话
<code>utf8.DecodeLastRuneInString(s)</code>	同上，但它输入的是字符串
<code>utf8.DecodeRune(b)</code>	返回 <code>b</code> 中的第一个 <code>rune</code> 和它占用的字节数，或者 <code>U+FFFD</code> （Unicode 替换字符？）和 0，如果 <code>b</code> 开始 <code>rune</code> 是非法的话
<code>utf8.DecodeRuneInString(s)</code>	同上，但它输入的是字符串
<code>utf8.EncodeRune(b, c)</code>	将 <code>c</code> 作为一个 UTF-8 字符并返回写入的字节数（ <code>b</code> 必须有足够的存储空间）
<code>utf8.FullRune(b)</code>	如果 <code>b</code> 的第一个 <code>rune</code> 是 UTF-8 编码的话，返回 <code>true</code>

续表

语法	含义/结果
<code>utf8.FullRuneInString(b)</code>	如果 <i>s</i> 的第一个 rune 是 UTF-8 编码的话, 返回 <code>true</code>
<code>utf8.RuneCount(b)</code>	返回 <i>b</i> 中的 rune 个数, 如果存在非 ASCII 字符的话这个值可能小于 <code>len(s)</code>
<code>utf8.RuneCountInString(s)</code>	同上, 但它输入的是字符串
<code>utf8.RuneLen(c)</code>	对 <i>c</i> 进行编码需要的字节数
<code>utf8.RuneStart(x)</code>	如果 <i>x</i> 可以作为一个 rune 的第一个字节的话, 返回 <code>true</code>
<code>utf8.Valid(b)</code>	如果 <i>b</i> 中的字节能正确表示一个 UTF-8 字符串, 返回 <code>true</code>
<code>utf8.ValidString(s)</code>	如果 <i>s</i> 中的字节能正确表示一个 UTF-8 编码的字符串, 返回 <code>true</code>

3.6.4 unicode包

`unicode`包主要提供了一些用来检查Unicode码点是否符合主要标准的函数, 例如, 判断一个字符是否是一个数字或者小写字母。表3-11列出了一些常用的函数。除了`unicode.ToLower()`和`unicode.IsUpper()`等, 还有一个通用的函数 `unicode.Is()`, 检查一个字符是否属于一个特定的Unicode分类。

```
fmt.Println(IsHexDigit('8'), IsHexDigit('x'), IsHexDigit('X'),
            IsHexDigit('b'), IsHexDigit('B'))
```

```
true·false·false·true·true
```

表3-11 unicode包

变量 *c* 是一个 `rune` 类型变量，表示一个 Unicode 码点。

语法	含义/结果
<code>unicode.Is(table, c)</code>	如果 <i>c</i> 在 <i>table</i> 中，返回 <code>true</code>
<code>unicode.IsControl(c)</code>	如果 <i>c</i> 是一个控制字符，返回 <code>true</code>
<code>unicode.IsDigit(c)</code>	如果 <i>c</i> 是一个十进制数字，返回 <code>true</code>
<code>unicode.IsGraphic(c)</code>	如果 <i>c</i> 是一个“图形”字符，如字母、数字、标记、符号或者空格返回 <code>true</code>
<code>unicode.IsLetter(c)</code>	如果 <i>c</i> 是一个字母，返回 <code>true</code>
<code>unicode.IsLower(c)</code>	如果 <i>c</i> 是一个小写字母，返回 <code>true</code>
<code>unicode.IsMark(c)</code>	如果 <i>c</i> 是一个标记，返回 <code>true</code>
<code>unicode.IsOneOf(tables, c)</code>	如果 <i>c</i> 在 <i>tables</i> 中的任何一个 <i>table</i> 中，返回 <code>true</code>
<code>unicode.IsPrint(c)</code>	如果 <i>c</i> 是一个可打印字符，返回 <code>true</code>
<code>unicode.IsPunct(c)</code>	如果 <i>c</i> 是一个标点符号，返回 <code>true</code>

续表

语法	含义/结果
<code>unicode.IsSpace(c)</code>	如果 <i>c</i> 是一个空格，返回 <code>true</code>
<code>unicode.IsSymbol(c)</code>	如果 <i>c</i> 是一个符号，返回 <code>true</code>
<code>unicode.IsTitle(c)</code>	如果 <i>c</i> 是一个标题大写字符，返回 <code>true</code>
<code>unicode.IsUpper(c)</code>	如果 <i>c</i> 是一个大写字母，返回 <code>true</code>
<code>unicode.SimpleFold(c)</code>	在与 <i>c</i> 的码点等价的码点集中，该方法返回最小的大于等于 <i>c</i> 的码点，否则如果不存在与其等价的码点，则返回最小的大于等于 0 的码点
<code>unicode.To(case, c)</code>	字符 <i>c</i> 的 <i>case</i> 版本，其中 <i>case</i> 可以是 <code>unicode.LowerCase</code> 、 <code>unicode.TitleCase</code> 或者 <code>unicode.UpperCase</code>
<code>unicode.ToLower(c)</code>	字母 <i>c</i> 的小写形式
<code>unicode.ToTitle(c)</code>	字符 <i>c</i> 的标题形式
<code>unicode.ToUpper(c)</code>	字母 <i>c</i> 的大写形式

`unicode` 包里有 `unicode.IsDigit()` 这样的函数，可以用来检查一个字符是否是一个十进制数字，但是并没有类似的函数可以检查十六进制数，所以这里用了一个自己实现的 `IsHexDigit()` 函数。

```
func IsHexDigit(char rune) bool {  
    return unicode.Is(unicode.ASCII_Hex_Digit, char)  
}
```

这个函数很简单，只用了一个 `unicode.Is()` 函数检查给定的字符是否在 `unicode.ASCII_Hex_Digit` 范围内，以此来判断这是否是一个十六进制数。我们还可以创建类似的函数来测试其他Unicode字符。

3.6.5 regexp包

这一节的表很多，主要是列举了`regexp`包里的函数和支持的正则表达式语法，还包含一些示例。在开始讲这一节之前，我们假设大家都有一定的正则表达式基础 [3]。

`regexp`包是Russ Cox的RE2正则表达式引擎的Go语言实现 [4]。这个引擎非常快而且是线程安全的。RE2引擎并不使用回溯，所以能够保证线性的执行时间 $O(n)$ ， n 是匹配字符串的长度，那些使用回溯的引擎的时间复杂度很容易达到指数级别 $O(2^n)$ （参见3.3 节的大 O 表示法）。获取出色性能的代价是不支持搜索时的反向引用，不过通常只要合理利用`regexp`的API就能绕开这些限制。

表 3-12 列出了 `regexp` 包里的函数，有 4 个可以创建一个 `*regexp.Regexp` 类型的值，表3-18和表3-19列出了`*regexp.Regexp`提供的方法。RE2引擎支持表3-13列出的转义序列、表3-14列出的字符类别、表3-15列出的零宽断言、表3-16列出的数量匹配，还有表3-17列出的标识。

`regexp.Regexp.ReplaceAll()` 方法和 `regexp.Regexp.ReplaceAllString()` 方法都支持按编号或者名字进行替换。编号对应于正则表达式中的括号括起来的捕获组，而名字则对应已命名的捕获组。尽管我们可以直接使用数字或名字引用来进行替换，例如`$2` 或者 `$filename` 等，但最好将数字和名字用大括号括起来，如`${2}`和`${filename}`等，如果替换的字符串中包含`$`字符，要使用`$$`来进行转义。

表3-12 `regexp`包函数列表

变量 *p* 和 *s* 都是字符串类型，*p* 表示正则匹配的模式。

语法	含义/结果
<code>regexp.Match(p, b)</code>	如果 []byte 类型的 <i>b</i> 和模式 <i>p</i> 匹配，返回 true 和 nil
<code>regexp.MatchReader(p, r)</code>	如果从 <i>r</i> 中读取的数据和模式 <i>p</i> 匹配，返回 true 和 nil， <i>r</i> 是一个 io.RuneReader
<code>regexp.MatchString(p, s)</code>	如果 <i>s</i> 和模式 <i>p</i> 匹配，返回 true 和 nil
<code>regexp.QuoteMeta(s)</code>	用引号安全地括起来的与正则表达式元字符相匹配的字符串
<code>regexp.Compile(p)</code>	如果模式 <i>p</i> 编译成功，返回一个 *regexp.Regexp 和 nil，参见表 3-18 和表 3-19
<code>regexp.CompilePOSIX(p)</code>	如果模式 <i>p</i> 编译成功，返回一个 *regexp.Regexp 和 nil，参见表 3-18 和表 3-19
<code>regexp.MustCompile(p)</code>	如果模式 <i>p</i> 编译成功返回一个 *regexp.Regexp，否则发生异常，参考表 3-18 和表 3-19
<code>regexp.MustCompilePOSIX(p)</code>	如果模式 <i>p</i> 编译成功返回一个 *regexp.Regexp，否则发生异常，参考表 3-18 和表 3-19

表3-13 regexp包支持的转义符号

语法	含义/结果
<code>\c</code>	原生字符 <i>c</i> ，例如 * 表示 * 是一个原生字符而不是一个量词
<code>\000</code>	表示一个八进制的码点
<code>\xHH</code>	表示指定的两个数字是十六进制
<code>\x{HHHH}</code>	表示给定的 1~6 个数字是十六进制的
<code>\a</code>	ASCII 码的响铃字符，等于 \007
<code>\f</code>	ASCII 码的换页符，等于 \014

续表

语法	含义/结果
<code>\n</code>	ASCII 码的换行符，等于 \012
<code>\r</code>	ASCII 码的回车符，等于 \015
<code>\t</code>	ASCII 码的制表符，等于 \011
<code>\v</code>	ASCII 码的垂直制表符，等于 \013
<code>\Q...\E</code>	原生匹配...中的所有字符即使它包含 *

表3-14 regexp包支持的字符类

语法	含义
[<i>chars</i>]	<i>chars</i> 中的任何字符
[[^] <i>chars</i>]	任何不在 <i>chars</i> 中的字符
[<i>:name:</i>]	任何在 <i>name</i> 字符类中的 ASCII 字符
	[[<i>:alnum:</i>]]≡[0-9A-Za-z] [[<i>:lower:</i>]]≡[a-z]
	[[<i>:alpha:</i>]]≡[A-Za-z] [[<i>:print:</i>]]≡[-~]
	[[<i>:ascii:</i>]]≡[\x00-\x7F] [[<i>:punct:</i>]]≡[!-/:-@[-'\{-~}
	[[<i>:blank:</i>]]≡[\t] [[<i>:space:</i>]]≡[\t\n\v\f\r]
	[[<i>:cntrl:</i>]]≡[\x00-\x1F\x7F] [[<i>:upper:</i>]]≡[A-Z]
	[[<i>:digit:</i>]]≡[0-9] [[<i>:word:</i>]]≡[0-9A-Za-z_]
	[[<i>:graph:</i>]]≡[!-~] [[<i>:xdigit:</i>]]≡[0-9A-Fa-z]
[[^] <i>:name:</i>]	任何不在 <i>name</i> 字符类中的 ASCII 字符
.	任何字符(如果指定 <i>s</i> 标识的话, 还包括换行符)
\d	任何 ASCII 码数字: [0-9]
\D	任何非数字的 ASCII 码: [[^] 0-9]
\s	任何 ASCII 码的空白字符: [\t\n\f\r]
\S	任何 ASCII 码的非空白字符: [[^] \t\n\f\r]
\w	任何 ASCII 码的单词字符: [0-9A-Za-z_]
\W	任何 ASCII 码的非单词字符: [[^] 0-9A-Za-z_]
\pN	任何一个在 N 指定的字符类里的 Unicode 字符, N 是一个单字母字符类, 例如\pL 匹配一个 Unicode 字母
\PN	任何一个不在 N 指定的字符类里的 Unicode 字符, N 是一个单字母字符类, 例如\PL 匹配所有非 Unicode 字母的字符
\p{Name}	任何在 Name 指定的字符类里的 Unicode 字符, 例如\p{Ll} 将匹配小写字母, \p{Lu} 匹配大写字母, \p{Greek} 匹配一个希腊字符
\P{Name}	任何不在 Name 字符类里的 Unicode 字符

表3-15 regexp包的零宽断言

语法	含义/结果
^	文本开始处 (如果 <i>m</i> 标识指定的话, 表示行首)
\$	文本末尾处 (如果 <i>m</i> 标识指定的话, 表示行尾)
\A	文本开始处
\Z	文本结尾处
\b	单词标界 (\W 和 \w 之间的字符, 或者 \A 和 \Z 之间的字符, 反过来也行)
\B	不是一个单词标界

表3-16 regexp包的数量匹配

语法	含义
<code>e?</code> or <code>e{0,1}</code>	贪婪匹配: <code>e</code> 出现 0 次或者 1 次
<code>e +</code> or <code>e{1,}</code>	贪婪匹配: <code>e</code> 出现 1 次或者多次
<code>e*</code> or <code>e{0,}</code>	贪婪匹配: <code>e</code> 出现 0 次或者多次
<code>e{m,}</code>	贪婪匹配: <code>e</code> 至少出现 <code>m</code> 次
<code>e{,n}</code>	贪婪匹配: <code>e</code> 最多出现 <code>n</code> 次
<code>e{m,n}</code>	贪婪匹配: <code>e</code> 最多出现 <code>n</code> 次, 最少出现 <code>m</code> 次
<code>e{m}</code> or <code>e{m}?</code>	<code>e</code> 只出现 <code>m</code> 次
<code>e??</code> or <code>e{0,1}?</code>	惰性匹配: <code>e</code> 出现 0 次或者 1 次
<code>e+?</code> or <code>e{1,}?</code>	惰性匹配: <code>e</code> 出现 1 次或者多次
<code>e*?</code> or <code>e{0,}?</code>	惰性匹配: <code>e</code> 出现 0 次或者多次
<code>e{m,}?</code>	惰性匹配: <code>e</code> 至少出现 <code>m</code> 次
<code>e{,n}?</code>	惰性匹配: <code>e</code> 最多出现 <code>n</code> 次
<code>e{m,n}?</code>	惰性匹配: <code>e</code> 最少出现 <code>m</code> 次, 最多出现 <code>n</code> 次

表3-17 `regex`包的标识和分组

语法	含义
<code>i</code>	匹配是大小写不敏感的（默认是区分大小写的）
<code>m</code>	开启多行模式，使 <code>^</code> 和 <code>\$</code> 能在每一行进行匹配（默认是单向模式的）
<code>s</code>	使 <code>.</code> 能够匹配换行符（默认 <code>.</code> 是不匹配换行符的）
<code>U</code>	将贪婪匹配和惰性匹配进行反转（例如通常量词后面带 <code>?</code> 表示惰性匹配，指定 <code>U</code> 之后，这种规则将表示贪婪匹配，而原表示贪婪匹配的将表示惰性匹配）
<code>(?flags)</code>	<code>flags</code> 标记从这一点开始生效（在 <code>flags</code> 标记前面加上 <code>-</code> 符号表示相反）
<code>(?flags:e)</code>	将给定的 <code>flags</code> 标记作用于表达式 <code>e</code> （在 <code>flags</code> 标记前面加上 <code>-</code> 号符表示相反）
<code>(e)</code>	表达式 <code>e</code> 的组和捕获
<code>(?P<name>e)</code>	表达式 <code>e</code> 的组和捕获，并显示的使用 <code>name</code> 来命名
<code>(?:e)</code>	表达式 <code>e</code> 的组但不包括捕获

表3-18 `*regex.Regexp`类型的方法 #1

`rx` 是 `*regexp.Regexp` 类型的变量, `s` 是用以匹配的字符串, `b` 是用以匹配的字节切片, `r` 是用以匹配的 `io.RuneReader` 类型变量, 还有 `n` 是最大匹配的次数(-1 表示不做限制), 返回 `nil` 的话表示没有匹配成功。

语法	含义/结果
<code>rx.Expand(...)</code>	由 <code>ReplaceAll()</code> 方法执行\$替换, 很少直接使用 (高级用法)
<code>rx.ExpandString(...)</code>	由 <code>ReplaceAllString()</code> 方法执行\$替换, 很少直接使用 (高级用法)
<code>rx.Find(b)</code>	使用最左匹配策略返回一个 <code>[]byte</code> 类型的切片或者 <code>nil</code>
<code>rx.FindAll(b, n)</code>	返回所有非重叠匹配的 <code>[] []byte</code> 类型切片或者 <code>nil</code>
<code>rx.FindAllIndex(b, n)</code>	返回一个 <code>[] []int</code> 类型的切片 (每一个元素是一个包含 2 项的切片), 其中每一个元素标识一个匹配或者 <code>nil</code> 。 例如 <code>b[pos[0]:pos[1]]</code> , 其中 <code>pos</code> 就是一个包含 2 项的切片
<code>rx.FindAllString(s, n)</code>	返回 <code>[] string</code> 类型的非重叠匹配或者 <code>nil</code>
<code>rx.FindAllStringIndex(s, n)</code>	返回一个 <code>[] []int</code> 类型的切片 (每一个元素是一个包含 2 项的切片), 其中每一个元素标识一个匹配或者 <code>nil</code> 。 例如 <code>s[pos[0]:pos[1]]</code> , 其中 <code>pos</code> 就是一个包含 2 项的切片
<code>rx.FindAllStringSubmatch(s, n)</code>	返回一个 <code>[] []string</code> 类型的切片 (一个字符串切片的切片, 其中每个字符串对应一个捕获) 或者 <code>nil</code>
<code>rx.FindAllStringSubmatchIndex(s, n)</code>	返回一个 <code>[] []int</code> 类型的切片 (每一个元素为包含 2 项的 <code>int</code> 类型切片, 每个元素对应一个匹配)
<code>rx.FindAllSubmatch(b, n)</code>	返回一个类型为 <code>[] [] []byte</code> 的三维切片 (该切片的元素是一个切片, 其中每一个切片又是一个 <code>[]byte</code> 类型的切片, 其中每一个切片对应一个捕获) 或者 <code>nil</code>
<code>rx.FindAllSubmatchIndex(b, n)</code>	返回一个类型为 <code>[] []int</code> 的二维切片 (一个其元素为包含 2 项的 <code>int</code> 类型切片, 每个元素对应一个匹配) 或者 <code>nil</code>
<code>rx.FindIndex(b)</code>	返回一个每个元素含有 2 项的 <code>[]int</code> 类型切片, 每个元素对应一个最左匹配或者 <code>nil</code> 。例如 <code>b[pos[0]:pos[1]]</code> , 其中 <code>pos</code> 是一个包含 2 项的切片
<code>rx.FindReaderIndex(r)</code>	返回一个每个元素含有 2 项的 <code>[]int</code> 类型切片, 每个元素对应一个最左匹配或者 <code>nil</code>

续表

语法	含义/结果
<code>rx.FindReaderSubmatchIndex(r)</code>	返回一个 <code>[]int</code> 类型的切片或者 <code>nil</code> ，对应最左匹配和捕获
<code>rx.FindString(s)</code>	返回一个最左匹配值或者空字符串
<code>rx.FindStringIndex(s)</code>	返回一个每个元素含有 2 项的 <code>[]int</code> 类型切片，每个元素对应一个最左匹配或者 <code>nil</code>
<code>rx.FindStringSubmatch(s)</code>	返回一个 <code>[]string</code> 类型的切片或者 <code>nil</code> ，对应最左匹配和捕获
<code>rx.FindStringSubmatchIndex(s)</code>	返回一个 <code>[]int</code> 类型切片或者 <code>nil</code> ，对应最左匹配和捕获

表3-19 `*regexp.Regexp`类型的方法 #2

`rx` 是 `*regexp.Regexp` 类型的变量，`s` 是用以匹配的字符串，`b` 是用以匹配的字节切片。

语法	含义/结果
<code>rx.FindSubmatch(b)</code>	返回最左的匹配或者捕获或者 <code>nil</code>
<code>rx.FindSubmatchIndex(b)</code>	返回最左匹配或者捕获的索引或者 <code>nil</code>
<code>rx.LiteralPrefix()</code>	返回所有匹配共有的原生前缀，和一个布尔变量（表明原生前缀能否匹配整个正则表达式）
<code>rx.Match(b)</code>	如果正则表达式匹配 <code>b</code> ，返回 <code>true</code>
<code>rx.MatchReader(r)</code>	同 <code>rx.Match()</code> ，但是从 <code>io.RuneReader</code> 里读取待匹配的数据
<code>rx.MatchString(s)</code>	同 <code>rx.Match()</code> ，但是匹配字符串 <code>s</code>
<code>rx.NumSubexp()</code>	返回正则表达式中有多少括起来的组（子表达式）
<code>rx.ReplaceAll(b, br)</code>	返回一个 <code>[]byte</code> 类型的 <code>b</code> 的副本，其中 <code>b</code> 中被匹配的部分都使用 <code>[]byte</code> 类型的 <code>br</code> 进行 <code>\$</code> 置换（见文中）
<code>rx.ReplaceAllFunc(b, f)</code>	返回一个 <code>[]byte</code> 类型的 <code>b</code> 的副本，其中 <code>b</code> 中被匹配的部分都使用函数 <code>f</code> 的返回值来替代， <code>f</code> 的原型为 <code>func([]byte) []byte</code> ，其参数为一个匹配项
<code>rx.ReplaceAllLiteral(b, br)</code>	返回一个 <code>[]byte</code> 类型的 <code>b</code> 的副本，其中 <code>b</code> 中被匹配的部分都使用 <code>[]byte</code> 类型的 <code>br</code> 进行替换
<code>rx.ReplaceAllLiteralString(s, sr)</code>	返回一个字符串类型的 <code>s</code> 的副本，其中 <code>s</code> 中被匹配的部分都使用字符串类型的 <code>sr</code> 进行替换
<code>rx.ReplaceAllString(s, sr)</code>	返回一个字符串类型的 <code>s</code> 的副本，其中 <code>s</code> 中被匹配的部分都使用字符串类型的 <code>sr</code> 进行 <code>\$</code> 置换

续表

语法	含义/结果
<code>rx.ReplaceAllStringFunc(s, f)</code>	返回一个字符串类型的 <i>s</i> 的副本, 其中 <i>s</i> 中被匹配的部分都使用函数 <i>f</i> 的返回值来替代, <i>f</i> 的原型为 <code>func(string) string</code> , 其参数为一个匹配项 <i>s</i>
<code>rx.String()</code>	返回正则表达式的字符串表示形式
<code>rx.SubexpNames()</code>	返回一个字符串 (不能用于修改目的), 包含所有已命名的字符类子表达式

一个典型的替换例子就是, 假如我们有一个形式如“**forname1 ... fornameN surname**”格式的名字列表, 现在我们把它们转换成“**surname, forname1 ... fornameN**”。看看我们是如何使用**regexp**包来实现这个功能且正确处理重音符号和其他的非英文字符。

```
nameRx := regexp.MustCompile('(\pL+\.(?:\s+\pL+\.?)*)\s+(\pL+)')
for i := 0; i < len(names); i++ {
    names[i] = nameRx.ReplaceAllString(names[i], " ${2}, ${1} ")
}
```

变量**names**是一个字符串切片, 保存了原来的名字列表。循环结束后**names**变量将被更新为修改后的名字列表。

这个正则表达式匹配一个或多个用空白分隔开的名字, 每个名字由一个或者多个**Unicode**字母组成 (名字后面可能有句号), 然后紧接着空白和姓, 姓也由一个或者多个**Unicode**字母组成。

根据数字编号来替换可能引入后期代码维护问题, 例如, 如果我们在中间插入一个捕获组, 则至少有一个数字是错误的。解决的办法就是使用显式命名的方式执行替换, 而不是依赖于数字型顺序。

```
nameRx := regexp.MustCompile(
    '(?P<forenames>\pL+\.(?:\s+\pL+\.?)*)\s+(?P<surname>\pL+)')
for i := 0; i < len(names); i++ {
    names[i] = nameRx.ReplaceAllString(names[i], " ${surname},
    ${forenames} ")
}
```

```
}
```

这里我们给两个捕获组指定了有意义的名字，使得正则表达式和替换字符串更容易被理解。

在Python或者Perl里，如果要匹配一个重复的单词，可以这样写“\b(w+)\s+\1\b”，但是这种正则语法需要依赖于反向引用，而这个恰好是Go语言里regexp引擎所不支持的，为了实现相同的效果，我们还得多写点代码才行。

```
wordRx := regexp.MustCompile("\w+')
if matches := wordRx.FindAllString(text, -1); matches != nil {
    previous := ""
    for _, match := range matches {
        if match == previous {
            fmt.Println(" Duplicate word: ", match)
        }
        previous = match
    }
}
```

这个正则表达式贪婪匹配一个或者多个单词，函数`regexp.Regexp.FindAllString()`返回一个不重叠的匹配结果，为`[]string`类型。如果至少存在一个匹配（`matches`不为`nil`），我们就遍历这个字符串切片，通过比较当前的单词和上一个单词，打印出所有重复的单词。

另一个常用的正则表达式是用来匹配一个配置文件里的“键:值”行，下面是一个例子，匹配指定的行并将其填充到`map`里面去。

```
valueForKey := make(map[string]string)
keyValueRx := regexp.MustCompile("\s*([[:alpha:]]\w*)\s*:\s*(.+)')
```

```

    if matches := keyValueRx.FindAllStringSubmatch(lines, -1); matches
    != nil {
        for _, match := range matches {
            valueForKey[match[1]] = strings.TrimRight(match[2], " \t ")
        }
    }
}

```

这个正则表达式是说跳过所有字符串开始处的空白，然后匹配一个键，键必须是以英文字母开头后面可接着0个或者多个字母、数字、下划线，然后是冒号和值，注意键和冒号之间或者值和冒号之间可以允许存在空白，值可以是任何字符但不包括换行符和字符串结束符。这里顺便提及，我们可以使用更短一点的[A-Za-z]替换[:alpha:]，或者如果我们想支持Unicode编码的键的话，可以使用(\pL[\pL\p{Nd}_]*), 表示一个Unicode字母后面紧接着0个或者多个Unicode字母、数字或者下划线。因为.+ 表达式不能匹配换行符，所以这个正则表达式能够处理连续包含多个“键:值”的字符串。

得益于贪婪匹配（默认），这个正则表达式能够除掉所有在值之前的空白。但我们必须使用裁剪函数除掉在值后面的空白，因为.+ 表达式的贪婪性意味着在其后跟随\s*将无效。我们也无法使用惰性匹配（例如.+?），因为这样的话只会匹配值的第一个单词，实际情况是值可能包含多个由空白分隔开的单词。

使用regexp.Regexp.FindAllStringSubmatch() 函数我们可以获得一个字符串切片的切片（[][]string）或者nil，-1表示尽可能多的匹配（不能重叠）。在我们这个例子里，每一个匹配都会产生包含3个字符串的切片，第一个字符串包含整个匹配，第二个字符串为键，第三个字符串为值。键和值都必须至少有一个字符，因为它们的最小数量是1。

尽管使用Go语言提供的xml.Decoder包来分析XML是最好的方法，但有时候我们只是简单地想得到XML文件里的属性值，格式通常为

name= " value " 或者name='value' 这样的字符串，这种情况下，用一个简单的正则表达式更加高效。

```
attrValueRx := regexp.MustCompile(regexp.QuoteMeta(attrName) + '='  
(?: " ([^ " ]+) " |'([^\']+)'))  
if indexes := attrValueRx.FindAllStringSubmatchIndex(attrs, -1);  
indexes != nil {  
    for _, positions := range indexes {  
        start, end := positions[2], positions[3]  
        if start == -1 {  
            start, end = positions[4], positions[5]  
        }  
        fmt.Printf( " '%s'\n " , attrs[start:end])  
    }  
}
```

attrValueRx 表达式匹配一个已经被转义了属性名后面紧随着一个等号和一个单双引号括起来的字符串。为 "|" 线正常工作而添加的一对方括号也会捕获匹配的表达式，但因为我们不希望捕获引号，所以我们将这一对方括号设置为非捕获状态（(?:)）。为了展示它是如何完成的，我们遍历得到匹配的索引而不是实际匹配的字符串，在这个例子里有3对索引，第一对索引是整个匹配的，第二对索引是双引号值的，第三对索引是单引号值的。当然，实际上只有一个值会被匹配，其他两个值都是-1。

和刚才的例子一样，我们这里也是匹配字符串里所有不重叠的匹配，然后得到一个[][]int类型的索引位置（或者为 nil）。对于每一个int 类型的 positions 切片，完整的匹配是切片 attrs[positions[0]:positions[1]]。引号包含的字符串是 attrs[positions[2]:positions[3]] 或者 attrs[positions[4]:positions[5]]，这

取决于你配置文件里引号的类型，上面这段代码默认为我们的配置文件使用的是双引号，但如果不是的话（如`start == -1`），那它就读取单引号的位置。

之前我们见过怎么去写一个`SimplifyWhitespace()`函数（参见3.6.1节），下面的代码使用正则表达式和`strings.TrimSpace()`函数来完成同样的功能。

```
simplifyWhitespaceRx := regexp.MustCompile('[\s\p{Zl}\p{Zp}]+')
text
strings.TrimSpace(simplifyWhitespaceRx.ReplaceAllLiteralString(text,
" ))
```

这个正则表达式对于字符串开头的空白只是做简单的跳过处理，对于结尾处的空白则使用`strings.TrimSpace()`函数来处理，这两部分的组合并没有做太多工作。函数`regexp.Regexp.ReplaceAllLiteralString()`将字符串中所有的匹配都给替换掉（`regexp.Regexp.ReplaceAllString()`和`regexp.Regexp.ReplaceAllLiteralString()`不同的是前者会对\$标识的变量进行展开，但后者不会）。所以，现在这种情况是，任何一个或多个的空白字符（ASCII编码的空格和Unicode行以及段落分隔符）都被替换成一个简单的空格。

下面是我们最后一个关于正则表达式的例子，我们来看看如何使用一个函数来执行具体的替换操作。

```
unaccentedLatin1Rx := regexp.MustCompile(
    '[ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝàáâãääåæçèéêëìíîïðò
    óôõöøùúûüýÿ]+')
unaccented := unaccentedLatin1Rx.ReplaceAllStringFunc(latin1,
UnaccentedLatin1)
```

这个正则表达式简单地匹配一个或者多个重音拉丁字母。只要存在一个匹配，`regexp.Regexp.ReplaceAllStringFunc()`函数都会调用作为第

二个参数传入的函数（函数必须是 `func (string) string` 类型的），这个函数接受匹配的字符串，该匹配的字符串将被该函数返回的字符串（可以是一个空字符串）替代。

```
func UnaccentedLatin1(s string) string {
    chars := make([]rune, 0, len(s))
    for _, char := range s {
        switch char {
            case 'À', 'Á', 'Â', 'Ã', 'Ä', 'Å':
                char = 'A'
            case 'Æ':
                chars = append(chars, 'A')
                char = 'E'
            //...
            case 'ý', 'ÿ':
                char = 'y'
        }
        chars = append(chars, char)
    }
    return string(chars)
}
```

这个函数简单地将所有重音的拉丁字符替换成它们的非重音形式，也会将连字 `æ`（在一些语言里这是一个全字符）替换为 `a` 和 `e`。当然，这个例子有些刻意为之，因为这里为执行转换其实我们只需写成 `unaccented := UnaccentedLatin1(latin1)`。

现在我们完成了对正则表达式例子的介绍。注意在表3-18和表3-19中，每个处理对象为字符串的函数都有一个对应处理对象为 `[]bytes` 上的

函数。书中还有一些其他例子也用到了正则表达式（例如1.6节和7.2.4.1节）。

现在我们已将Go语言的strings和相关的包都介绍完了，我们将用一个使用了一些Go语言string函数的例子来结束整章所学的内容，后面照常会有一些练习。

3.7 例子: m3u2pls

这一节我们介绍一个短小精悍的程序，它从命令行输入读取任意后缀名为.m3u的音乐播放列表文件并输出一个等同的.pls播放列表文件。程序里使用了很多strings包里的函数，还有一些这两章接触过的东西，同时还会介绍一些新的东西。

下面是一个.m3u文件解开的内容，中间一大部分用省略号替代了。

```
#EXTM3U
#EXTINF:315,David Bowie - Space Oddity
Music/David Bowie/Singles 1/01-Space Oddity.ogg
#EXTINF:-1,David Bowie - Changes
Music/David Bowie/Singles 1/02-Changes.ogg
...
#EXTINF:251,David Bowie - Day In Day Out
Music/David Bowie/Singles 2/18-Day In Day Out.ogg
```

文件的开始内容是一个字符串常量 #EXTM3U。每一首歌用两行来表示。第一行以字符串#EXTINF:开始，紧跟着是歌曲的持续时间（以秒为单位），然后是一个逗号，接着就是歌曲名。如果持续时间是-1的话意味着歌曲的长度是未知的（或者是未知格式）。第二行是保存

歌曲的路径。这里我们使用开源且非专利保护的音频压缩格式并采用Ogg封装格式（www.vorbis.com），以及Unix风格的路径分隔符。

下面是一个等同的.pls文件的释放内容，同样使用省略号省略歌曲的大部分。

```
[playlist]
File1=Music/David Bowie/Singles 1/01-Space Oddity.ogg
Title1=David Bowie - Space Oddity
Length1=315
File2=Music/David Bowie/Singles 1/02-Changes.ogg
Title2=David Bowie - Changes
Length2=-1
...
File33=Music/David Bowie/Singles 2/18-Day In Day Out.ogg
Title33=David Bowie - Day In Day Out
Length33=251
NumberOfEntries=33
Version=2
```

.pls文件格式相比.m3u格式稍微可读一点，文件以字符串[playlist]开始，每一首歌用3个“键/值”条目分别来表示文件名、标题和持续时间（以秒为单位）。实际上.pls文件格式相当于是一种特殊的.ini文件（Windows系统的配置文件格式），在ini里每一个键（在一个中括号表示的节里面）必须是唯一的，因此我们用数字来进行区分。最后文件以两行元数据结束。

m3u2pls程序（在文件m3u2pls/m3u2pls.go里）在运行时需要在命令行提供一个后缀名为.m3u的文件，他会将一个对应的.pls文件写到标准输出（即控制台）。我们可以使用重定向将.pls数据输出到一个实际的文件。下面是这个程序用法的一个例子。

```
$/m3u2pls Bowie-Singles.m3u > Bowie-Singles.pls
```

这里我们让程序从 **Bowie-Singles.m3u** 文件读取数据，然后利用控制台的重定向功能将 **.pls** 版本格式的数据写到 **Bowie-Singles.pls** 文件里（当然，如果你能用其他方式来转换，那就更好了，正好这也是我们这一节后面的练习所要求的）。

后面我们会介绍差不多整个程序的代码，除了一些 **import** 语句。

```
func main() {  
    if len(os.Args) == 1 || !strings.HasSuffix(os.Args[1], ".m3u ") {  
        fmt.Printf( " usage: %s <file.m3u>\n ", filepath.Base(os.Args[0]))  
        os.Exit(1)  
    }  
    if rawBytes, err := ioutil.ReadFile(os.Args[1]); err != nil {  
        log.Fatal(err)  
    } else {  
        songs := readM3uPlaylist(string(rawBytes))  
        writePlsPlaylist(songs)  
    }  
}
```

main() 函数首先会检查命令行是否指定了包含 **.m3u** 后缀名的文件。函数 **strings.HasSuffix()** 输入两个字符串，如果第一个字符串是以第二个字符串结束的话返回 **true**。如果没有指定 **.m3u** 文件的话就打印使用帮助信息并退出程序。函数 **filepath.Base()** 返回给定路径的基名（例如文件名），还有 **os.Exit()** 函数会在退出前清理所有的资源，例如，停止所有的 **goroutine** 和关闭所有打开的文件，然后将它的参数返回给操作系统。

如果我们从命令行读取到一个 `.m3u` 文件，我们就尝试用 `ioutil.ReadFile()` 函数将整个文件的数据读取出来，这个函数返回文件的所有字节流（用 `[]byte` 类型保存）和一个 `error` 变量。如果读取过程中没发生任何错误的话 `error` 的值为 `nil`，否则（例如文件不存在或者不可读），我们用 `log.Fatal()` 函数往控制台（实际上是 `os.Stderr`）输出错误信息，然后以退出码1退出整个程序。

如果我们成功读取了一个文件，我们将原始的字节流转换成字符串，这里假定这些字节均表示一个7位的ASCII码或者UTF-8编码的Unicode字符，然后立即将这个字符串作为参数传递给自定义函数 `readM3uPlaylist()`，这个函数返回一个 `Song` 切片（`[]Song` 类型），然后我们用函数 `writePlsPlaylist()` 将这些歌曲写到标准输出。

```
type Song struct {
    Title      string
    Filename string
    Seconds int
}
```

这里我们定义了一个 `Song` 类型的结构体（关于结构体的说明参见6.4节），方便用来单独保存和文件格式无关的歌曲信息。

```
func readM3uPlaylist(data string) (songs []Song) {
    var song Song
    for _, line := range strings.Split(data, "\n") {
        line = strings.TrimSpace(line)
        if line == "" || strings.HasPrefix(line, "#EXTM3U") {
            continue
        }
        if strings.HasPrefix(line, "#EXTINF:") {
            song.Title, song.Seconds = parseExtinfLine(line)
        }
    }
}
```

```

    } else {
        song.Filename = strings.Map(mapPlatformDirSeparator, line)
    }
    if song.Filename != "" && song.Title != "" &&
song.Seconds != 0 {
        songs = append(songs, song)
        song = Song{}
    }
}
return songs
}

```

函数以字符串的形式传入整个.m3u文件的内容，然后返回一个从字符串中分析出来的包含所有歌曲信息的切片。刚开始程序声明了一个空的Song类型变量，叫song，得益于Go语言总是将变量初始化为零值，song的初始内容为两个空字符串且song.Seconds的值为0。

函数的核心是一个for...range循环（参见5.3节），strings.Split()函数用来将整个包含.m3u内容的字符串按行化分，然后利用for来遍历每一行，如果有一行为空或者是第一行（例如，这行内容是以“#EXTM3U”开始的），就转到continue声明处，这就简单地将控制流返回到for循环强制执行下一次遍历，或者如果没有其他行了的话就结束循环。

如果行是以字符串“#EXTINF:”开始的，就将这一行传给parseExtinfLine()函数做分析，这个函数返回一个字符串和一个int型值，并立即赋值给当前song的Title和Seconds字段，否则，它就假定这一行包含当前song的文件名（全路径）。

我们并不直接保存文件名，而是借助 strings.Map()函数，它调用自定义函数mapPlatformDirSeparator()将行中的目录分隔符转换成程序所在平台的本地格式，再将结果字符串保存在当前 song的Song.Filename

里。函数 `strings.Map()` 传入一个签名为 `func(rune) rune` 的映射函数和一个字符串，对字符串中的每一个字符，调用传入的映射函数并将这个字符作为参数传入，同时该字符将被映射函数返回的字符替换掉。当然，这和我们之前讲过的是一样的。按照Go语言的惯例，一个 `rune` 表示的字符它的值是一个Unicode码点。

如果当前 `song` 的文件名和标题都不为空，而且歌曲的持续时间不为0，当前的 `song` 就会被追加到返回值 `songs` 里（`[]Song` 类型），同时通过将空的一个 `Song` 结构赋值给它，将当前 `song` 设置为零值（两个空的字符串和0）。

```
func parseExtinfLine(line string) (title string, seconds int) {
    if i := strings.IndexAny(line, "-0123456789 "); i > -1 {
        const separator = " , "
        line = line[i:]
        if j := strings.Index(line, separator); j > -1 {
            title = line[j+len(separator):]
            var err error
            if seconds, err = strconv.Atoi(line[:j]); err != nil {
                log.Printf(" failed to read the duration for '%s': %v\n", title, err)
                seconds = -1
            }
        }
    }
    return title, seconds
}
```

这个函数用来分析 `#EXTINF:duration,title` 这种形式的文本行，其中 `duration` 是一个整数，其值可以为-1或者大于0。

`strings.IndexAny()`函数用来查找第一个数字或者负号的位置。如果索引的位置为-1的话说明没有找到，其他值则表示函数第二个参数指定的字符串中的任意一个字符第一次出现的位置，这时变量*i*保存了歌曲持续时间的第一个数字（或者是-）的位置。

一旦我们知道数字从哪里开始，我们就将行从数字开始的地方进行切分，这样很容易地就把字符串行首的“#EXFINFO:”抛弃掉，现在这行字符串的形式就成了 `duration,title`。

第二个if语句使用`strings.Index()`函数获得“, ”字符串在行内第一次出现的索引位置，如果返回值为-1则表明逗号不存在。

`title`是从逗号后面到行结束之间的文本，要从逗号后面开始切分，我们需要知道逗号开始的位置*j*和逗号占用的字节数`len(separator)`。当然，我们知道一个逗号是7位的ASCII码字符，所以它的长度是1，但我们这里显示的方法可以工作在任何Unicode字符上，不管该字符使用多少个字节表示。

`duration`是一个数，它从文本起始处开始但不包括第*j*个字符（逗号所在的地方）。我们使用 `strconv.Atoi()`函数将这个数转换成 `int` 型的值，如果转换失败了我们简单地设置持续时间为-1，也就是一个“未知的持续时间”，同时将这个问题记录到日志，这样用户就能察觉到它。

```
func mapPlatformDirSeparator(char rune) rune {  
    if char == '/' || char == '\\' {  
        return filepath.Separator  
    }  
    return char  
}
```

对于文件名的每一个字符，（在 `readM3uPlaylist()` 函数里）`strings.Map()`都会调用这个函数。它将文件名中的路径分隔符替换成特

定平台的目录分隔符，对于其他字符则原样返回。

像大多数跨平台的编程语言和库一样，Go内部对所有的平台都使用Unix风格的目录分隔符，即便是Windows也如此。但是，对用户可见的输出或者人类可读的文件数据，我们推荐使用平台指定的目录分隔符。我们可以使用 `filepath.Separator` 常量来实现这个功能，在Unix类系统上它的值是“/”，在Windows平台上则是“\”。

在这个例子里我们不知道我们所读取的路径使用的是“/”还是“\”，所以我们对两种都做了处理。不过，如果我们为了确信一个路径用的是否是“/”，我们可以对它调用`filepath.FromSlash()`函数：在Unix类系统上返回的结果没有变化，但是在Windows系统上它会将“/”替换成“\”。

```
func writePlsPlaylist(songs []Song) {
    fmt.Println( " [playlist] " )
    for i, song := range songs {
        i++
        fmt.Printf( " File%d=%s\n " , i, song.Filename)
        fmt.Printf( " Title%d=%s\n " , i, song.Title)
        fmt.Printf( " Length%d=%d\n " , i, song.Seconds)
    }
    fmt.Printf( " NumberOfEntries=%d\nVersion=2\n " , len(songs))
}
```

这个函数以.pls的格式往os.Stdout（例如控制台）输出songs的数据，所以如果需要输出到文件的话就一定要使用文件重定向。文件首先输出节的头部（ " [playlist] " ），然后对于每首歌曲在它自己的行内输出歌曲的文件名、标题和持续时间（单位为秒），最后输出两行元数据。

3.8 练习

这一章有两个练习题，第一个是修改之前的命令程序，第二个需要从头创建一个Web应用程序，但这个可选的。

(1) 上一节的m3u2pls程序能很好地完成.m3u播放列表格式到.pls格式之间的转换，但如果它也能执行反向转换的话这个程序就更有用了。所以，第一道题的要求就是复制m3u2pls 文件夹到目标文件夹，比如 my_playlist 文件夹，然后创建一个新的程序叫playlist，来实现这个新功能，程序的用法应该是： playlist, <file.[pls|m3u]>。

如果这个文件调用时指定的是一个.m3u文件，它做的事情跟m3u2pls是一样的：将.pls格式的文件数据写到控制台。但如果这个程序调用时指定的是.pls文件，它应该将.m3u格式的文件数据写到控制台。参考答案在playlist/playlist.go文件里，大概增加了50行左右的代码。

(2) 与人名有关的数据清理、匹配和挖掘程序按发音而非拼写进行人名匹配通常可以产生更好的结果。有很多算法可以用来将名字匹配到英文名，但最古老最简单的算法是Soundex。

经典的Soundex算法能生成一个大写字母后跟3个数字的soundex值。例如，根据大部分的Soundex算法“Robert”和“Rupert”这两个名字都有相同的soundex值“R163”，但是对于名字“Ashcroft”和“Ashcraft”，一些Soundex算法（包括本练习答案中的一种算法）产生的值是“A226”，而另一些算法却是“A261”。

第二道题的要求就是写一个Web应用，主要是两个页面，第一个页面（路径是/）要能显示一个简单的表单，通过让用户输入一个或者多个名字然后查看它们的soundex值，如图3-3左图所示。第二个页面（路径是/test）能够执行这个程序的soundex()函数来处理一个字符串列表，将每个结果和我们期望值进行比较，如图3-3右图所示。

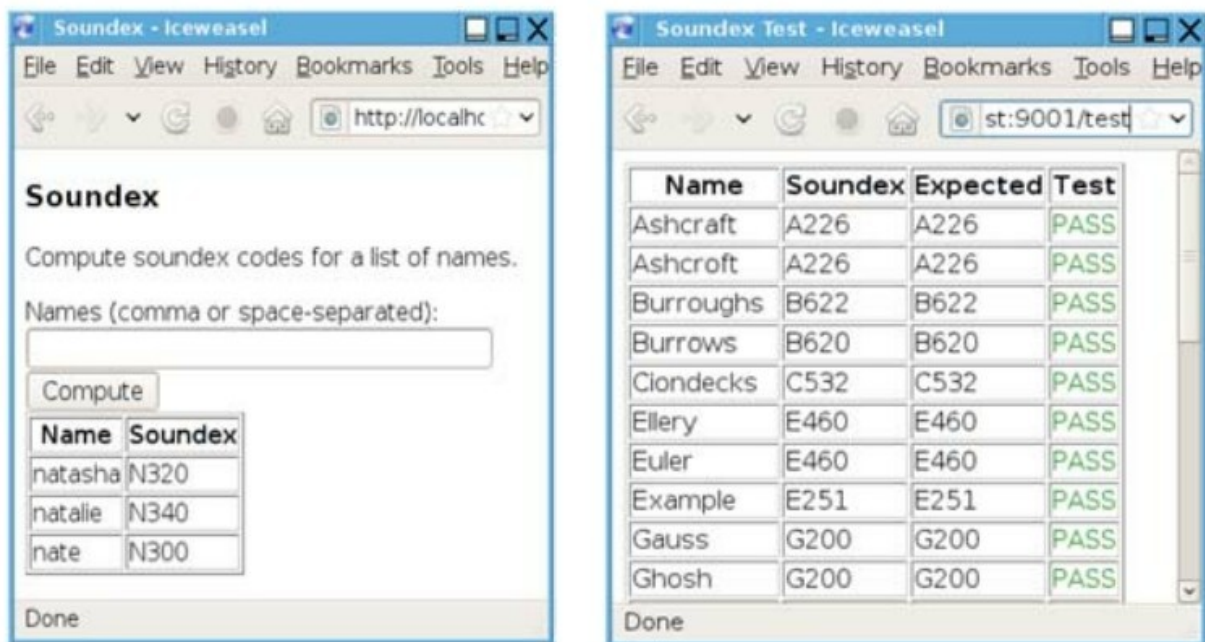


图3-3 Linux上的Soundex应用

希望能够快速开始的读者可以复制之前完成的Web 应用程序（`statistics`、`statistics_ans`、`quadratic_ans1`、`quadratic_ans2`）以让这个程序的主干运行起来，然后把重心放在实现soundex和test页面的功能上。

参考答案在`soundex/soundex.go`文件里，大概150行代码，其中`soundex()`函数本身有20行代码，不过它很巧妙地使用`[]int`来建立一个大写字母和数字之间的映射关系。这个答案所用的算法基于Rosetta Code网站上（rosettacode.org/wiki/Soundex）的Python实现，和那个网站的Go语言版本或者维基百科（en.wikipedia.org/wiki/Soundex）上的实现所产生的结果略有不同。测试数据放在`soundex/soundex-test-data.txt`文件里。

自然而然地，读者可以自由地实现任何一种自己钟情的算法，或者甚至实现一个更加高级的算法，例如某种Metaphone算法，并简单地调整一下测试数据。

[1].Go语言也有两个内置的打印函数，即 `print()`与 `println()`。通常不推荐使用，它们的存在只是为了方便 Go编译器的实现者，可能会被删除。

[2].C、C++和Python 2程序员会发现Go语言的格式字符串非常熟悉，但存在一些细微的差别。例如。Go语言的%d可以用于任何整数，无论它的大小如何和有无符号。

[3].有一本非常好的讲解正则表达式的书，书名是Mastering Regular Expressions（《精通正则表达式》），见附录C。本书作者的另一本书 Progamming in Python 3，也有一章是讲解Python正则表达式的（正则表达式语法的一个子集）。这一章的内容可以从网页 www.informit.com/title/9780321680563免费下载（点击“Sample Content”链接并下载第13章即可）。

[4].关于RE2的信息，包括讲述了它的合理性、性能及实现的文档链接可以从code.google.com/p/re2获得。

第4章 集合类型

本章第一节首先介绍了Go语言中的值、指针以及引用类型，因为理解这些概念对于本章的后续节以及本书的后续章节都是必要的。Go语言的指针与C和C++中的指针类似，无论是语法上还是语意上。但是Go语言的指针不支持指针运算，这样就消除了C和C++程序中一类潜在的bug。Go语言也不用free()函数或者delete操作符，因为Go语言有垃圾回收器，并且自动管理内存 [1]。Go语言引用类型的值以一种独特而简单的方式创建，并且一旦创建后就可以像Java或者Python中的对象引用一样使用。Go语言的值的工作方式与其他大多数主流语言一致。

本章的其他节将深入讲解Go语言内置的集合类型。其中包含了Go语言的所有内置类型：数组、切片和映射。这些类型功能齐全并且高效，能够满足大部分需求。标准库中也提供了一些额外的更加特别的集合类型container/heap、container/list和container/ring。这些类型可能在某些特殊情况下更高效。后续章节中有些关于使用堆和列表的小程序（参见9.4.3节）。第6章有个例子，展示了如何创建一个平衡二叉树（参见6.5.3节）。

4.1 值、指针和引用类型

本节我们讨论变量持有什么内容（值、指针以及指向数组、切片和映射的引用），并在接下来的节中讨论如何使用数组、切片和映射。

通常情况下 Go 语言的变量持有相应的值。也就是说，我们可以将一个变量想像成它所持有的值来使用。其中有些例外是对于通道、函数、方法、映射以及切片的引用变量，它们持有的都是引用，也即保存指针的变量。

值在传递给函数或者方法的时候会被复制一次。这对于布尔变量或者数值类型来说是非常廉价的，因为每个这样的变量只占1~8个字节。按值传递字符串也非常廉价，因为Go语言中的字符串是不可变的，Go语言编译器会将传递过程进行安全的优化，因此无论传递的字符串长度多少，实际传递的数据量都会非常小。（每个字符串的代价在64位的机器上大概是16字节，在32位的机器上大概是8字节 [2] ）。当然，如果修改了一个传入的字符串（例如，使用 += 操作符），Go语言必须创建一个新的字符串，并且复制原始的字符串并将其加到该字符串之后，这对于大字符串来说很可能代价非常大。

与C和C++ 不同，Go语言中的数组是按值传递的，因此传递一个大数组的代价非常大。幸运的是，在Go语言中数组不常用到，因为我们可以使用切片来代替。我们将在下面章节讲解切片的用法。传递一个切片的成本与字符串差不多（在64位机器上为16字节，在32位机器上为12字节），无论该切片的长度或者容量是多大 [3] 。另外，修改切片也不会导致写时复制的负担，因为不同于字符串的是，切片是可变的（如果一个切片被修改，这些修改对于其他所有指向该切片的引用变量都是可见的）。

图4-1说明了变量及它们所占用内存空间的关系。在图中，内存地址以灰色显示，因为它们是可变的，而粗体则表示变化。

语句	变量	值	类型	内存地址
<code>y := 1.5</code>	<code>y</code>	1.5	float64	0xf8400000f8
<code>y++</code>	<code>y</code>	2.5	float64	0xf8400000f8
<code>z := math.Ceil(y)</code>	<code>y</code>	2.5	float64	0xf8400000f8
	<code>x</code>	2.5	float64	Ceil() 中 y 的可修改的副本
	<code>z</code>	3.0	float64	0xf8400000c0

图4-1 简单类型值在内存中的表示

从概念上讲，变量是赋给一内存块的名字，该内存块用于保存特定的数据类型。因此如果我们进行一个短声明 `y := 1.5`，Go语言就会分配一个足够放置一个 **float64** 数的内存块（8个字节）并将数字 **1.5** 保存到该内存块中。之后只要 `y` 还保存在作用域中，Go语言就会将变量 `y` 等同于这个内存块。因此如果我们在声明语句后面跟上一条 `y++` 语句，Go语言将修改变量 `y` 对应的内存块中保存的数值。然而如果我们将 `y` 传递给一个函数或者方法，Go语言就会传递一个 `y` 的副本。从另一方面来讲，Go语言会创建另一个与所调用的函数的参数名相关联的变量，并将 `y` 的值复制到为该新变量对应的新内存块中。

有时我们需要一个函数修改我们传入的值。由于值类型是复制的，因此任何修改只作用于其副本，而其原始值将保持不变。同时，传值的成本也可能非常高，因为它们会很大（例如，一个数组或者一个包含许多字段的结构体）。此外，本地变量在不再使用时会被垃圾回收（当它们不再被引用或者不在作用域范围时），然而在许多情况下我们希望自己来管理变量的生命周期而非由它们的作用域决定。

通过使用指针，我们可以让参数的传递成本最低并且内容可修改，而且还可以让变量的生命周期独立于作用域。指针是指保存了另一个变量内存地址的变量。创建的指针是用来指向另一个某种类型的变量，这样就保证了 Go语言知道该指针所指向的值占用多大的空间。我们马上会看到，一个被指针指向的变量可以通过该指针来修改。不

管指针所指向值的大小，指针的传递是非常廉价的（64位的机器上占8字节，32位的机器上占4字节）。同时，对于某个指针所指向的变量，只要保证至少有一个指针指向该变量，该变量就会在内存中保存足够长的时间，因此它们的生命周期独立于我们所创建的作用域。[\[4\]](#)

在Go语言中&操作符有多重用处。当用作二元操作符时，它是按位与操作。当用作一元操作符时，它返回的是操作数的地址，该地址可由一个指针保存。在图4-2的第三个语句中，我们将int型变量 x 的内存地址赋值给类型为*int的变量pi（指向int型变量的指针）。一元操作符 & 有时也被称为取址操作符。正如图4-2中的箭头所示，术语“指针”也描述了一个事实，即保存了另一变量内存地址的变量通常被认为是“指向”了那个变量。

语句	变量	值	类型	内存地址
x := 3	x	3	int	0xf840000148
y := 22	y	22	int	0xf840000150
x == 3 && y == 22				
pi := &x	x	3	int	0xf840000148
	pi	0xf840000148	*int	0xf840000158
*pi == 3 && x == 3 && y == 22				
x++	x	4	int	0xf840000148
	pi	0xf840000148	*int	0xf840000158
*pi == 4 && x == 4 && y == 22				
*pi++	x	5	int	0xf840000148
	pi	0xf840000148	*int	0xf840000158
*pi == 5 && x == 5 && y == 22				
pi := &y	y	22	int	0xf840000150
	pi	0xf840000150	*int	0xf840000158
*pi == 22 && x == 5 && y == 22				
*pi++	y	23	int	0xf840000150
	pi	0xf840000150	*int	0xf840000158
*pi == 23 && x == 5 && y == 23				

图4-2 指针和值

同样，*操作符也有多重用处。当用作二元操作符时，它将其操作数相乘。而当用作一元操作符时，它返回它所作用的指针所指向变量的值。因此，在图4-2中，pi := &x语句之后*pi和x可以相互交换着使用（但当pi被赋值给另一个变量的指针后就不行了）。并且，由于它们与同一块内存地址相关联，任何作用于其中一个变量的改变都会改变另一个。一元操作符*有时也叫做内容操作符、间接操作符或者解引用操作符。

图4-2说明了如果我们将指针所指向变量的值改变，其值如我们所预期的那样改变，并且当我们将该指针解引用时（`*pi`），它返回修改后的新值。我们也可以通过指针来改变其值。例如，`*pi++`意味着将指针所指的值得增加；当然，这只有在其类型支持++操作符时才能够通过编译，比如Go语言内置的数值类型。

一个指针不必始终指向同一个值。例如，从图4-2的底部开始，我们将一个指针指向了不同的值（`pi := &y`），然后通过指针来改变其值。我们可以轻易地直接改变y的值（使用`y++`），然后使用`*pi`来返回y的新值。

指针也可以指向另一个指针（或者指向指针的指针的指针）。使用指向值的指针叫做间接引用。如果我们使用指向指针的指针，这就叫做使用多重间接引用。这在C和C++中非常普遍，但在Go语言中不常用到，因为Go语言使用引用类型。这里有个简单的例子。

```
z := 37                // z的类型为int
pi := &z               // pi的类型为 *int （指向int型的指针）
ppi := &pi             // ppi的类型为 **int （指向int类型指针的指针）

fmt.Println(z, *pi, **ppi)
**ppi++               // 语意上等同于(*(*ppi))++和*(*ppi)++
fmt.Println(z, *pi, **ppi)

37 37 37
38 38 38
```

在上面的代码片段中，`pi`是一个`*int`类型（指向`int`类型的指针）的指针，它指向一个`int`类型的变量`z`，同时`ppi`是一个指向`pi`的`**int`类型的指针（指向`int`类型指针的指针）。当解引用时，对于每一层的间接引用我们使用`*`操作符，因此`*ppi`解引用`ppi`变量产生一个`*int`，即一个内存地址，再次应用`*`操作符（`**ppi`）时，我们得到所指向的整型值。

除了当做乘法和解引用操作符之外，*操作符也可以当做类型修改符。当一个*置于类型名的左边时，它将原来声明一个特定类型的值的语义修改为了声明一个指向特定类型值的指针。这在图4-2的“类型”一栏中展示。

让我们用一个小例子来解释下目前为止所讨论的内容。

```
i := 9
j := 5
product := 0
swapAndProduct1(&i, &j, &product)
fmt.Println(i, j, product)
```

```
5 9 45
```

这里我们创建了 3 个类型为整型的变量，并给它们一个初始值。然后我们调用自定义的`swapAndProduct1()`函数。该函数接收3个整型变量指针，保证指针指向的头两个整型数按递增顺序排列，并且让第三个指针指向的整型数赋值为前两个整型数的乘积。由于该函数接受指针而非值类型的参数，我们必须传入指向`int`类型值的指针，而非该`int`类型值。每当我们看到取址操作符`&`被用于函数调用时，我们都要假设对应的变量值可能在函数内被修改。下面是该`swapAndProduct1()`函数的实现。

```
func swapAndProduct1(x, y, product *int) {
    if *x > *y {
        *x, *y = *y, *x
    }
    *product = *x * *y // 编译器也能处理这样的写法：
    *product=*x**y
}
```

函数的参数声明`*int`使用`*`类型修改符来声明其参数全是指向整型数的指针。当然，这也意味着我们只能传入指向整型变量的指针（使用取址操作符`&`），而非传入整型变量或者整型数。

在函数内部，我们更关心指针所指向的值，因此我们从头到尾都使用解引用操作符`*`。在最后一个可执行的行中，我们将两个指针所指向的值乘起来，然后将其结果赋值给另一个指针所指向的变量。当有两个连续的`*`出现时，Go语言会根据上下文将其识别成乘法而非两个解引用。在函数内部，指针是`x`、`y`和`product`，但是在函数调用处，它们所指向的值为3个整型变量`i`、`j`和`product`。

在C和早期的C++代码中，用这种方式实现函数是非常普遍的现象，但在Go语言中这种写法不是必须的。如果我们只有一个或者不多的几个值，在Go语言中更符合常规的做法是直接返回它们，而如果有许多值要传递的话，以切片或者映射（我们马上会看到，无需指针也可以非常廉价地传递它们）的形式传递就可以了，如果它们的类型不一致则将其放在一个结构体中再用指针传递。这里有个没用到指针的更简单的改进版。

```
i := 9
j := 5
i, j, product := swapAndProduct2(i, j)
fmt.Println(i, j, product)
```

5 9 45

这里是我们所写的对应的`swapAndProduct2()`函数。

```
func swapAndProduct2(x, y int) (int, int, int) {
    if x > y {
        x, y = y, x
    }
}
```

```
    return x, y, x * y
}
```

这个版本的函数应该比第一个版本清晰多了，但没用指针也导致了该函数不能就地交换数据。

在C和C++中，函数参数包含一个布尔类型指针来表示成功或者失败的做法是很常见的。这在Go语言中可以通过在函数签名处包含一个*bool变量来实现，但直接以最后一个返回值的形式返回一个布尔型的成功标志（或者最好是一个error值）的写法更好用，这也是Go语言的推荐做法。

在目前为止已经展示的代码片段中，我们使用取址操作符&来取得函数参数或者本地变量的地址。Go语言的自动内存管理机制使得这样做非常安全，因为只要一个指针引用一个变量，那个变量就会在内存中得以保留。这也是为什么在Go语言的函数内部返回指向本地变量的指针是安全的（在C/C++中，对于非静态变量的同样操作将是灾难）。

在某些场景下，我们需要传递非引用类型的可修改值，或者需要高效地传入大类型的值，这个时候我们需要用到指针。Go语言提供了两种创建变量的语法，同时获得指向它们的指针。其中一种方法是使用内置的new()函数，另一种方法是使用地址操作符。为了比较一下，我们将介绍这两种语法，并用两种语法分别创建一个扁平结构的结构体类型值。

```
type composer struct{
    name      string
    birthYear int
}
```

给定这个结构体定义，我们可以创建 composer 值或指向 composer 值的指针，即*composer类型的变量。在这两种情况下，我们都可以利用Go语言对结构体初始化的支持使用大括号来初始化数据。

```

    antonio := composer{ " António Teixeira " , 1707}    // composer类型
值
    agnes := new(composer)                                // 指向
composer的指针
    agnes.name, agnes.birthYear = " Agnes Zimmermann " , 1845
    julia := &composer{}                                  // 指向
composer的指针
    julia.name, julia.birthYear = " Julia Ward Howe " , 1819
    augusta := &composer{ " Augusta Holmès " , 1847}      // 指向
composer的指针
    fmt.Println(antonio)
    fmt.Println(agnes, augusta, julia)
    {António Teixeira 1707}
    &{Agnes Zimmermann 1845} &{Augusta Holmès 1847} &{Julia
Ward Howe 1819}

```

当 Go语言打印指向结构体的指针时，它会打印解引用后的结构体内容，但会将取址操作符&作为前缀来表示它是一个指针。上面创建了 agnes和julia两个指针的代码片段用于解释以下两种用法的等同性，只要其类型可以使用大括号进行初始化：

new(Type) =&Type{ }

这两种语法都分配了一个Type类型的空值，同时返回一个指向该值的指针。如果Type不是一个可以使用大括号初始化的类型，我们只可以使用内置的new()函数。当然，我们不必担心该值的生命周期或怎么将其删除，因为Go语言的内存管理系统会帮我们打理一切。

使用结构体的&Type{ }语法的一个好处是我们可以为其指定初始值，正如我们这里创建augusta指针时所做的那样（后面我们将看到，

我们也可以只声明一些可选的字段而将其他字段设为它们的0值，参见6.4节）。

除了值和指针之外，Go语言也有引用类型（另外Go语言还有接口类型，但在大多数实际使用中我们可以把接口看成某种类型的引用，引用类型将在本书稍后介绍，参见6.3节）。一个引用类型的变量指向内存中某个隐藏的值，它保存着实际的数据。保存引用类型的变量传递时也非常廉价（在64位机器上一个切片占16字节，一个映射占8字节），其使用语法与值一样（我们不必取得一个引用类型的地址，在需要得到该引用所指的值时也无需解引用它）。

一旦我们遇到需要在一个函数或方法中返回超过四五个值的情况时，如果这些值是同一类型的话最好使用一个切片来传递，如果其值类型各异则最好传递一个指向结构体的指针。传递一个切片或一个指向结构体的指针的成本都比较低，同时也允许我们修改数据。让我们用一些小例子来解释这些。

```
grades := []int{87, 55, 43, 71, 60, 43, 32, 19, 63}
inflate(grades, 3)
fmt.Println(grades)
```

```
[261 165 129 213 180 129 96 57 189]
```

这里我们在一个整型切片之上进行一个操作。映射和切片都是引用类型，并且映射或者切片项中的任何修改（无论是直接的还是在它们所传入的函数中间接的修改）对于引用它们的变量来说都是可见的。

```
func inflate(numbers []int, factor int) {
    for i := range numbers {
        numbers[i] *= factor
    }
}
```

```
}
```

`grades`切片作为参数`numbers`传入函数。但与传入值不同的是，任何作用于`numbers`的更改都会作用于`grades`，因为它们都指向同一个切片。

由于我们希望原地修改切片的值，所以使用了一个循环来轮流获得其中的值。我们没有使用`for index、item...range`这样的循环是因为这样只能得到其所操作的切片元素的副本，导致其副本与因数相乘之后将该值丢弃，而原始切片的值则保持不变。我们本来可以使用更熟悉的类似于其他语言的`for`循环（例如`for i := 0; i < len(numbers); i++`），但我们可以使用更为方便的`for index := range`语法（下一章会讲解所有的`for`循环语法，参见5.3节）。

我们假设有一个矩形类型，将一个矩形的位置保存为左上角和右下角的`x、y`坐标以及机器填充色。我们可以将该矩形的数据表示成一个结构体。

```
type rectangle struct {  
    x0, y0, x1, y1 int  
    fill           color.RGBA  
}
```

现在我们可以创建一个矩形类型的值，打印它的内容，调整大小，然后再打印它的内容。

```
rect := rectangle{4, 8, 20, color.RGBA{0xFF, 0, 0, 0xFF}}  
fmt.Println(rect)  
resizeRect(&rect, 5, 5)  
fmt.Println(rect)
```

```
{4 8 20 10 {255 0 0 255}}  
{4 8 25 15 {255 0 0 255}}
```

正如我们在前面章节所提到的，虽然 Go语言不认识我们所定义的矩形类型，但它还是能够用合适的格式将其打印出来。代码下面的输出清楚地显示出`resizeRect()`功能的正确性。与传入整个矩形（其中的整型至少占16字节）不同的是，我们只传入其地址（无论结构体多大，在64位系统中都是8字节）。

```
func resizeRect(rect *rectangle, Δwidth, Δheight int) {  
    (*rect).x1 += Δwidth    // 令人厌恶的显式解引用  
    rect.y1 += Δheight      // "." 操作符能够自动解引用结构体  
}
```

函数的第一个语句使用显式的解引用操作，展示了其底层发生的操作。`(*rect)`引用的是该指针所指出的矩形值，其中的`.x1`引用矩形的`x1`字段。第二个语句所给出的才是使用结构体值的常用方法。结构体指针也使用与第二个语句一样的语法，在这种情况下，需依赖 Go语言来为我们解引用。之所以这样是因为，Go语言的`.`（点）操作符能够自动地将指针解引用为它所指向的结构体 [5]。

Go语言中有些类型是引用类型：映射、切片、通道、函数和方法。与指针不同的是，引用类型没有特殊的语法，因为它们就像值一样。指针也可以指向一个引用类型，虽然它只对切片有用，但有时这个用法也很关键（我们将在下一章节中看到使用指向切片的指针的案例，参见5.7节）。

如果我们定义了一个变量来保存一个函数，该变量得到的实际是该函数的引用。函数引用知道它们所引用的函数的签名，因此不能传递一个签名不匹配的函数引用。这也消除了一些在某些语言中可能发生的非常麻烦的错误和崩溃，因为这些语言在使用函数指针时不保证这些函数的签名正确。我们已经看到了一些传入函数引用的例子，比如当我们传递一个映射函数给`strings.Map()`函数时。我们会在本书余下的部分看到更多使用指针和引用类型的例子。

4.2 数组和切片

Go语言的数组是一个定长的序列，其中的元素类型相同。多维数组可以简单地使用自身为数组的元素来创建。

数组的元素使用操作符[]来索引，索引从0开始。因此一个数组的首元素是array[0]，其最后元素是array[len(array)-1]。数组是可更改的，因此我们使用将array[index]放置在赋值操作符的左边这样的语法来设置index位置处的元素内容。我们也可以在赋值语句的右边或者一个函数调用中使用该语法，以获得该元素。

数组使用以下语法创建：

[length]Type

[N]Type{value1, value2,..., valueN}

[...]Type{value1, value2,..., valueN}

如果在这种场景中使用了...（省略符）操作符，Go语言会为我们自动计算数组的长度。（我们将在本章后面及第5章看到，这个省略操作符也可以用于其他目的。）在任何情况下，一个数组的长度都是固定的并且不可修改。

以下示例展示了如何创建和索引数组。

```
var buffer [20]byte
```

```
var grid1 [3][3]int
```

```
grid1[1][0], grid1[1][1], grid1[1][2] = 8, 6, 2
```

```
grid2 := [3][3]int{{4, 3}, {8, 6, 2}}
```

```
cities := [...]string{ " Shanghai " , " Mumbai " , " Istanbul " , "
Beijing " }
```

```
cities[len(cities)-1] = " Karachi "
```

```
fmt.Println( " Type Len Contents " )
```

```

fmt.Printf( " %-8T %2d %v\n " , buffer, len(buffer), buffer)
fmt.Printf( " %-8T %2d %q\n " , cities, len(cities), cities)
fmt.Printf( " %-8T %2d %v\n " , grid1, len(grid1), grid1)
fmt.Printf( " %-8T %2d %v\n " , grid2, len(grid2), grid2)

```

Type Len Contents

```

[20]uint8 20 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[4]string 4 [ " Shanghai " " Mumbai " " Istanbul " " Karachi " ]
[3][3]int 3[[000][862][000]]
[3][3]int 3[[430][862][000]]

```

正如上面的`buffer`、`grid1`和`grid2`变量所展示的，当创建数组时，如果没有被显式地初始化或者只是部分初始化，Go语言会保证数组的所有项都被初始化成其相应的零值。

数组的长度可以使用`len()`函数获得。由于数组的长度是固定的，因此它们的容量总是等于其长度，对于数组而言`cap()`函数和`len()`函数返回的数字一样。数组可以使用与字符串或者切片一样的语法进行切片，只是其结果为一个切片，而非数组。同时，就像字符串和切片一样，数组也可以使用`for...range`循环来进行迭代（参见5.3节）。

一般而言，Go语言的切片比数组更加灵活、强大且方便。数组是按值传递的（即传递副本，虽然可以通过传递指针来避免）而不管切片的长度和容量如何，传递成本都会比较小，因为它们是引用。（无论包含了多少个元素，一个切片在64位机器上是以16字节的值进行传递的，在32位机器上是以12字节的值进行传递的。）数组是定长的，而切片可以调整长度。Go语言标准库中的所有公开函数使用的都是切片而非数组 [6]。我们建议使用切片来代替数组，除非在特殊的案例下有非常特别的需求必须用数组。数组和切片都可以使用表4-1中所给出的语法来进行切片。

表4-1 切片操作

语法	含义/结果
<code>s[n]</code>	切片 <code>s</code> 中索引位置为 <code>n</code> 的项
<code>s[n:m]</code>	从切片 <code>s</code> 的索引位置 <code>n</code> 到 <code>m-1</code> 处所获得的切片
<code>s[n:]</code>	从切片 <code>s</code> 的索引位置 <code>n</code> 到 <code>len(s)-1</code> 处所获得的切片
<code>s[:m]</code>	从切片 <code>s</code> 的索引位置 0 到 <code>m-1</code> 处所获得的切片
<code>s[:]</code>	从切片 <code>s</code> 的索引位置 0 到 <code>len(s)-1</code> 处所获得的切片
<code>cap(s)</code>	切片 <code>s</code> 的容量；总是 $\geq \text{len}(s)$
<code>len(s)</code>	切片 <code>s</code> 中所包含项的个数；总是 $\leq \text{cap}(s)$
<code>s[:cap(s)]</code>	增加切片 <code>s</code> 的长度到其容量，如果两者不同的话

Go语言中的切片是长度可变、容量固定的相同类型元素的序列。我们将在后文看到，虽然切片的容量固定，但也可以通过将其切片来收缩或者使用内置的`append()`函数来增长。多维切片可以自然地使用类型为切片的元素来创建，并且多维切片内部的切片长度也可变。

虽然数组和切片所保存的元素类型相同，但在实际使用中并不受此限。这是因为其类型可以是一个接口。因此我们可以保存任意满足所声明的接口的元素（即它们定义了该接口所需的方法）。然后我们可以让一个数组或者切片为空接口`interface{}`，这意味着我们可以存储任意类型的元素，虽然这会导致我们在获取一个元素时需要使用类型断言或者类型转变，或者两者配合使用。（接口会在第6章讲到。反射的内容参见9.4.9节。）

我们可以使用以下语法创建切片：

```
make([]Type, length, capacity)
```

```
make([]Type, length)
```

```
[]Type{}
```

```
[]Type{value1, value2,..., valueN}
```

内置函数 `make()` 用于创建切片、映射和通道。当用于创建一个切片时，它会创建一个隐藏的初始化为零值的数组，然后返回一个引用该隐藏数组的切片。该隐藏的数组与 Go语言中的所有数组一样，都是固定长度的，如果使用第一种语法创建，那么其长度即为切片的容

量；如果使用第二种语法创建，那么其长度即为切片的长度；如果使用复合语法创建（第三种或者第四种），那么其长度为大括号中项的个数。

一个切片的容量即为其隐藏数组的长度，而其长度则为不超过该容量的任意值。在第一种语法中，切片的长度必须小于或者等于容量，虽然这种语法一般也是在我们希望其初始长度小于容量的时候才使用。第二种、第三种和第四种语法都是用于当我们希望其长度和容量相同时。复合语法（第四种）非常方便，因为它允许我们使用一些初始值来创建切片。

语法 `[]Type{}` 与语法 `make([]Type, 0)` 等价，两者都创建一个空切片。这并不是无用的，因为我们可以使用内置的 `append()` 函数来有效地增加切片的容量。然而在实际使用中如果我们需要一个初始化为空的切片，使用 `make()` 函数来创建会更实用，只需将长度设为 0，并且将容量设为一个我们估计该切片需要保存的元素个数。

一个切片的索引位置范围为从 0 到 `len(slice)-1`。一个切片可以再重新切片以减小长度，并且如果一个切片的长度小于其容量值，那么该切片也可以重新切片以将其长度增长为容量值。我们将在后文提到，可以使用内置的 `append()` 函数来增加切片的容量。

图4-3从概念的视角给出了切片与其隐藏数组的关系。下面这些是它所给出的切片。

```
s := []string{ " A ", " B ", " C ", " D ", " E ", " F ", " G " }
t := s[:5]           // [A B C D E]
u := s[3 : len(s)-1] // [D E F]
fmt.Println(s, t, u)
u[1] = " X "
fmt.Println(s, t, u)
```

[A B C D E F G] [A B C D E] [D E F]

```
[A B C D x F G] [A B C D x] [D x F]
```

由于切片s、t和u都是同一个底层数组的引用，其中一个改变会影响到其他所有指向该相同数组的任何其他引用。

```
s := new([7]string)[:]
```

```
s[0], s[1], s[2], s[3], s[4], s[5], s[6] = " A ", " B ", " C ", " D ", " E ", " F ", " G "
```

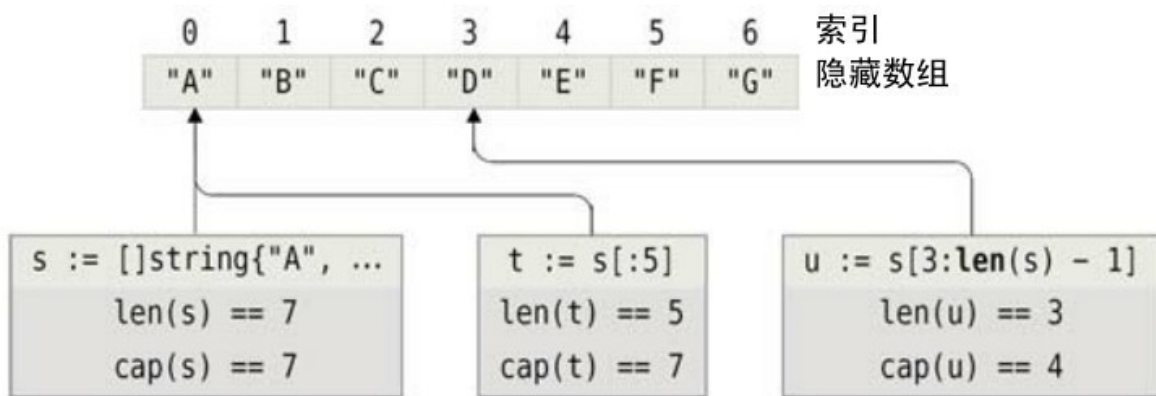


图4-3 切片与其底层数组的概念图

使用内置的**make()**函数或者复合语法是创建切片的最好方式，但是这里我们给出了一种在实际中不常用到但是能够很明显地说明数组及其切片之间关系的方法。第一条语句使用内置的**new()**函数创建一个指向数组的指针，然后立即取得该数组的切片。这会创建一个其长度和容量都与数组的长度相等的切片，但是所有元素都会被初始化为零值（在这里是空字符串）。第二条语句通过将每个元素设置成我们想要的初始值来完成整个切片的设置。设置完成之后，该切片与上文中使用复合语法创建的切片完全一样。

下面基于切片的例子除了我们将**buffer**的容量设为大于其长度以演示如何使用外，与我们之前所看到的基于数组的例子功能完全一致。

```
bufer := make([]byte, 20, 60)
```

```
grid1 := make([][]int, 3)
```

```

for i := range grid1 {
    grid1[i] = make([]int, 3)
}
grid1[1][0], grid1[1][1], grid1[1][2] = 8, 6, 2
grid2 := [][]int{{4, 3, 0}, {8, 6, 2}, {0, 0, 0}}
cities := []string{ " Shanghai " , " Mumbai " , " Istanbul " , " Beijing
" }
cities[len(cities)-1] = " Karachi "
fmt.Println( " Type Len Cap Contents " )
fmt.Printf( " %-8T %2d %3d %v\n " , buffer, len(buffer), cap(buffer),
buffer)
fmt.Printf( " %-8T %2d %3d %q\n " , cities, len(cities), cap(cities),
cities)
fmt.Printf( " %-8T %2d %3d %v\n " , grid1, len(grid1), cap(grid1),
grid1)
fmt.Printf( " %-8T %2d %3d %v\n " , grid2, len(grid2), cap(grid2),
grid2)

```

Type	Len	Cap	Contents
[]uint8	20	60	[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[]string	4	4	[" Shanghai " " Mumbai " " Istanbul " " Karachi "]
[][]int	3	3	[[0 0 0] [8 6 2] [0 0 0]]
[][]int	3	3	[[4 3 0] [8 6 2] [0 0 0]]

buffer的内容仅仅是前面 len(buffer)个项，除非我们将其重新切片，否则我们将无法取到其他元素。后面几节会讲到如何重新做切片。

我们使用初始值长度为3（即包含3个切片）和容量为3（由于无特殊说明的情况下默认初始容量的值为其长度）来创建一个切片的切片 `grid1`。然后我们为 `grid1` 最外层的切片的每一项设置成包含它们自身的切片，该包含的切片也含有3个项。自然地，我们也可以让最内层的切片长度不一。

对于 `grid2` 我们必须声明每一个值，因为使用了复合语法来创建它，而Go语言没有其他方法来知道我们需要多少个项。总之，我们创建了一个包含不同长度切片的切片，例如 `grid2 := [][]int{{9,7}, {8}, {4, 2, 6}}` 可以使得 `grid2` 是一个长度为3并且包含3个长度分别为2， 1和3的切片的切片。

4.2.1 索引与分割切片

一个切片是一个隐藏数组的引用，并且对于该切片的切片也引用同一个数组。这里有一个例子可以解释上面所提到的。

```
s := []string{ " A ", " B ", " C ", " D ", " E ", " F ", " G " }
```

```
t := s[2:6]
```

```
fmt.Println(t, s, " = ", s[:4], " + ", s[4:])
```

```
s[3] = " x "
```

```
t[len(t)-1] = " y "
```

```
fmt.Println(t, s, " = ", s[:4], " + ", s[4:])
```

```
[C D E F] [A B C D E F G] = [A B C D] + [E F G]
```

```
[C x E y] [A B C x E y G] = [A B C x] + [E y G]
```

我们可以改变数据，无论是通过原始切片 `s` 还是通过切片 `s` 的切片 `t`，它们的底层数据都改变了，因此两个切片都受影响。上面的代码片段也说明，对于一个切片 `s` 和一个索引值 `i` ($0 \leq i \leq \text{len}(s)$)，`s` 等于 `s[:i]` 与

`s[i:]`的连接。在前面章节中讲字符串引用的时候我们也看到了类似的相等性：

`s == s[:i] + s[i:]` // `s`是一个字符串，`i`是整型， $0 \leq i \leq \text{len}(s)$

图4-4 展示了切片`s`，包括它所有有效的索引位置和上面代码中用到的切片。任何切片中的第一个索引位置都为0，而最后一个则为`len(s) - 1`。

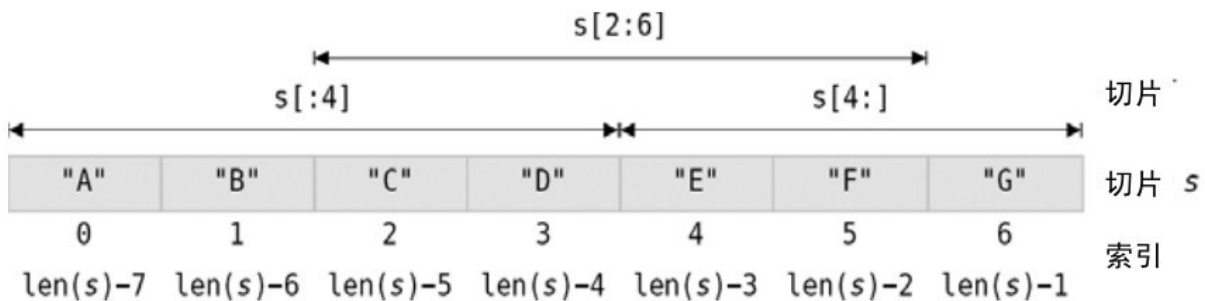


图4-4 切片剖析

与字符串不同的是，切片不支持`+`或者`+=`操作符，然而可以很容易地往切片后面追加以及插入和删除元素，我们随后将看到（参见4.2.3节）。

4.2.2 遍历切片

一个常用到的需求是遍历一个切片中的元素。如果我们想要取得切片中的某个元素而不想修改它，那我们可以使用`for...range`循环，如果想要修改它则可以使用带循环计数器的`for`循环。关于前者，这里有一个例子。

```
amounts := []float64{237.81, 261.87, 273.93, 279.99, 281.07, 303.17,
    231.47, 227.33, 209.23, 197.09}
sum := 0.0
for _, amount := range amounts {
    sum += amount
}
```



```
}  
fmt.Printf( " Σ %.1f → %.1f\n " , amounts, sum)
```

```
Σ[237.8 261.9 273.9 280.0 281.1 303.2 231.5 227.3 209.2  
197.1] → 2503.0
```

`for...range`循环首先初始化一个从0开始的循环计数器，在本例中我们使用空白符将该值丢弃（`_`），然后是从切片中复制对应元素。这种复制即使对于字符串来说也是代价很小的（因为它们按引用传递）。这意味着任何作用于该项的修改都只作用于该副本，而非切片中的项。

自然地，我们可以使用切片的方式来遍历切片中的一部分。例如，如果我们想要遍历切片的前5个元素，我们可以这样写 `for _, amount := range amounts[:5]`。

如果我们想修改切片中的项，我们必须使用可以提供有效切片索引而非仅仅是元素副本的`for`循环。

```
for i := range amounts {  
    amounts[i] *= 1.05  
    sum += amounts[i]  
}  
fmt.Printf( " Σ %.1f → %.1f\n " , amounts, sum)
```

```
Σ[249.7 275.0 287.6 294.0 295.1 318.3 243.0 238.7 219.7  
206.9] → 2628.1
```

这里我们将切片中的每个元素值增加了5%，并将其总和累加起来。

当然，切片也可以包含自定义类型的元素。以下是包含了一个自定义方法的自定义类型。

```
type Product struct {  
    name string  
    price float64  
}  
  
func (product Product) String() string{  
    return fmt.Sprintf( " %s (%.2f) " , product.name, product.price)  
}
```

这里将**Product**类型定义为一个包含一个字符串和一个**float64**类型字段的结构体。我们同时也定义了**String()**方法来控制Go语言如何使用**%v**格式符输出**Product**的内容（我们之前介绍了打印格式符，参见3.5节。在1.5节中我们简单地引入了自定义类型和方法。第6章将提供更多的内容。）

```
products := []*Product{{ " Spanner " , 3.99}, { " Wrench " , 2.49},  
    { " Screwdriver " , 1.99}}  
  
fmt.Println(products)  
  
for _, product := range products {  
    product.price += 0.50  
}  
  
fmt.Println(products)
```

```
[Spanner (3.99) Wrench (2.49) Screwdriver (1.99)]
```

```
[Spanner (4.49) Wrench (2.99) Screwdriver (2.49)]
```

这里我们创建了一个包含指向 **Product** 指针（**[]*Product**）的切片，然后立即使用 3 个***Product** 来将其初始化。之所以可以这样做是因为 Go语言足够灵活能够识别出来一个**[]*Product**需要的是指向**Product**的指针。我们这里所写的只是**products := []*Product{&Product{ " Spanner**

" , 3.99}, &Product{ " Wrench " , 2.49}, &Product{ " Screwdriver " , 1.99}}的简化版（在4.1节中，我们使用&Type{}来创建一个该类型的新值，并立即得到了一个指向它的指针）。

如果我们没有定义Product.String()方法，那么格式符%v（该格式符在fmt.Println()以及类似的函数中被显式地调用）输出的就是Product的内存地址，而非Product的内容。同时需注意的是Product.String()方法接收一个Product值，而非一个指针。然而这不成问题，因为Go语言足够聪明，当需要传值的地方传入的是一个指针的时候，Go会自动将其解引用 [7] 。

我们之前也提到for...range循环不能用于修改所遍历元素的值，然而在这里我们成功地将切片中所有产品的价格都增加了。在每一次遍历中，变量 product 被赋值为一个*Product副本，这是一个指向products所对应的底层Product的指针。因此，我们所做的修改是作用于指针所指向的值，而非*Product指针的副本。

4.2.3 修改切片

如果我们需要往切片中追加元素，可以使用内置的append()函数。这个函数接受一个需要被追加的切片，以及一个或者多个需要被追加的元素。如果我们希望往一个切片中追加另一个切片，那么我们必须使用...（省略号）操作符来告诉Go语言将被添加进来的切片当成多个元素。需要添加的元素类型必须与切片类型相同。以字符串为例，我们可以使用省略号语法将字节添加进一个字节类型切片中。

```
s := []string{ " A " , " B " , " C " , " D " , " E " , " F " , " G " }  
t := []string{ " K " , " L " , " M " , " N " }  
u := []string{ " m " , " n " , " o " , " p " , " q " , " r " }  
s = append(s, " h " , " i " , " j " ) // 添加单一的值
```

```

s = append(s, t...)           // 添加切片中的所有值
s = append(s, u[2:5]...)     //添加一个子切片
b := []byte{'U', 'V'}
letters := " WXY "
b = append(b, letters...)    //将一个字符串字节添加进一个字节切片中

```

```

fmt.Printf( " %v\n%s\n " , s, b)

```

```

[A B C D E F G h i j K L M N o p q]

```

```

UVwxy

```

内置的append()函数接受一个切片和一个或者更多个值，返回一个（可能为新的）切片，其中包含原始切片的内容并将给定的值作为其后续项。如果原始切片的容量足够容纳新的元素（即其长度加上新元素数量的和不超过切片的容量），append()函数将新的元素放入原始切片末尾的空位置，切片的长度随着元素的添加而增长。如果原始切片没有足够的容量，那么append()函数会隐式地创建一个新的切片，并将其原始切片的项复制进去，再在该切片的末尾加上新的项，然后将新的切片返回，因此需要将append()的返回值赋值给原始切片变量。

有时我们不仅想往切片的末尾插入项，也想往切片的前面或者中间插入项。下面这些例子使用了我们自定义的InsertStringSliceCopy()函数，它接收一个我们要插入切片的参数、一个用于插入的切片以及需插入切片的索引位置。

```

s := []string{ " M " , " N " , " O " , " P " , " Q " , " R " }
x := InsertStringSliceCopy(s, []string{ " a " , " b " , " c " }, 0) //At the
front
y := InsertStringSliceCopy(s, []string{ " x " , " y " }, 3) // In the
middle
z := InsertStringSliceCopy(s, []string{ " z " }, len(s)) // At the end

```

```
fmt.Printf( " %v\n%v\n%v\n%v\n " , s, x, y, z)
```

```
[M N O P Q R]
```

```
[a b c M N O P Q R]
```

```
[M N O x y P Q R]
```

```
[M N O P Q R z]
```

自定义的`InsertStringSliceCopy()`函数创建了一个新的切片（这也是为什么在输出的时候切片`s`没有被改变的原因），使用内置的`copy()`函数来复制第一个切片，并插入第二个切片。

```
func InsertStringSliceCopy(slice, insertion []string, index int)[]string{
    result := make([]string, len(slice)+len(insertion))
    at := copy(result, slice[:index])
    at += copy(result[at:], insertion)
    copy(result[at:], slice[index:])
    return result
}
```

内置的`copy()`函数接受两个包含相同类型元素的切片（这两个切片也可能是同一个切片的不同部分，包括重叠的部分）。该函数将第二个切片（源切片）的元素复制到第一个切片（目标切片），并返回所复制元素的数量。如果源切片为空，那么`copy()`函数将安全地什么都不做。如果目标切片的长度不够来容纳目标切片中的项，那么那些无法容纳的项将被忽略。如果目标切片的容量比其长度大，我们可以在复制之前通过语句`slice = slice[:cap(slice)]`来将其长度增加至容量值。

传入`copy()`函数的两个切片的类型必须相同（例外情况是当第一个切片（目标切片）的类型为`[]byte`时，第二个切片（源切片）可以是`[]byte`类型或者字符串类型）。如果源切片是一个字符串，那么会将其字节码拷入第一个参数。（关于这类用法的一个例子将在第6章6.3节讲解。）

在自定义的函数`InsertStringSliceCopy()`函数开始处，我们首先创建一个新的切片（`result`），其容量足够容纳所传入的两个切片的项。然后将第一个切片到索引位置处的子切片复制到`result`切片中。接下来我们将所需插入的切片复制到`result`切片的末尾，其位置从我们上次复制子切片时所到达的位置开始。然后再将第一个切片中剩下的子切片复制到`result`切片中，其位置也从我们上次复制子切片时所到达的位置开始。对于最后一次复制，函数`copy()`的返回值被忽略，因为我们不再需要它了。最后，返回`result`切片。

如果所传入的索引位置为0，那么第一条语句中的`slice[:index]`为`slice[:0]`，因此无需进行复制。类似地，如果所传入的索引位置大于等于切片的长度，则最后一条复制语句为`slice[len(slice):]`（即一个空切片），因此也无需复制。

以下这个函数的功能与之前的`InsertStringSliceCopy()`函数类似，但是更加简短。不同点在于，`InsertStringSlice()`函数会更改原始切片（也可能修改需插入的切片），然而`InsertStringCopy()`函数不会修改这些切片。

```
func InsertStringSlice(slice, insertion []string, index int) []string {  
    return append(slice[:index], append(insertion, slice[index:]...)...)  
}
```

`InsertStringSlice()`函数将原始切片从索引位置处到末尾处的子切片追加到插入切片中，并将得到的切片插入到原始切片从开始处到索引位置处的子切片末尾。其返回值为被叠加后的原始切片。（回忆下，`append()`函数接受一个切片和一个或者多个值，因此我们必须使用省略号语法来将一个切片转换成它的多个元素值，而本例中我们必须这样做两次。）

使用Go语言的标准切片语法可以将元素从切片的开头和末尾处删除，但是将其从中间删除就费点事。我们接下来首先看看如何在原地

从一个切片的头和尾删除一个元素，然后是从中间删除。之后再看看如何复制一个从原始切片删除了部分元素后得到的切片，但原始切片保持不变。

```
s := []string{ " A ", " B ", " C ", " D ", " E ", " F ", " G " }  
s = s[2:] // 从开始处删除s[:2]子切片  
fmt.Println(s)
```

```
[C D E F]
```

通过使用再切片，可以轻易地从一个切片的开头删除元素。

```
s := []string{ " A ", " B ", " C ", " D ", " E ", " F ", " G " }  
s = s[:4] // 从末尾处删除s[4:]子切片  
fmt.Println(s)
```

```
[A B C D]
```

从一个切片的末尾处删除元素使用的也是再切片方法，与从切片的开头处删除元素一样。

```
s := []string{ " A ", " B ", " C ", " D ", " E ", " F ", " G " }  
s = append(s[:1], s[5:]...) // 从中间删除s[1:5]  
fmt.Println(s)
```

```
[A F G]
```

从中间取得元素非常简单。例如，要取得切片 `s` 中间的3个元素，我们可以使用表达式 `s[2:5]`。但是要从切片的中间删除元素就略微需要一点点技巧。这里我们使用 `append()` 函数来将需要删除元素的前后两部分连接起来，并将其赋值回给 `s`。

明显地，使用`append()`函数并且将新切片赋值回给原始切片来删除元素这样的做法会改变原始切片。以下几个例子使用了自定义的`RemoveStringSliceCopy()`函数，它会返回原始切片的副本，但删除了其开始和结尾索引位置处之间的元素。

```
s := []string{ " A ", " B ", " C ", " D ", " E ", " F ", " G " }
x := RemoveStringSliceCopy(s, 0, 2)           // 从头部删除s[:2]
y := RemoveStringSliceCopy(s, 1, 5)           // 从中间删除s[1:5]
z := RemoveStringSliceCopy(s, 4, len(s))      // 从结尾处删除s[4:]
fmt.Printf( " %v\n%v\n%v\n%v\n ", s, x, y, z)
```

```
[A B C D E F G]
[C D E F G]
[A F G]
[A B C D]
```

由于`RemoveStringSliceCopy()`函数首先复制了原始切片的元素，因此原始切片保持完整。

```
func RemoveStringSliceCopy(slice []string, start, end int)[]string{
    result := make([]string, len(slice)-(end-start))
    at := copy(result, slice[:start])
    copy(result[at:], slice[end:])
    return result
}
```

在`RemoveStringSliceCopy()`函数中，我们首先创建一个新切片（`result`），并保证其容量足够大。然后我们将原始切片中从开头到`start`索引位置处的子切片拷进`result`切片中。接下来我们再将原始切片中从索引位置处到结尾处的子切片追加到`result`切片中。最后，我们返回`result`切片。

当然我们也可以创建一个更加简单的工作于原始切片的 `RemoveStringSlice()` 函数，而不是创建一个副本。

```
func RemoveStringSlice(slice []string, start, end int) []string{
    return append(slice[:start], slice[end:]...)
}
```

这是对之前所给出的使用 `append()` 函数从切片中间删除元素的例子的通用化修改。其返回的切片为原始切片中将从 `start` 位置处到（但不包括）`end` 位置处的项删除后的切片。

4.2.4 排序和搜索切片

标准库中的 `sort` 包提供了对整型、浮点型和字符串类型切片进行排序的函数，检查一个切片是否排序好的函数，以及使用二分搜索算法在一个有序切片中搜索一个元素的函数。同时提供了通用的 `sort.Sort()` 和 `sort.Search()` 函数，可用于任何自定义的数据。这些函数在表4-2中有列出。

表4-2 `sort`包中的函数

语法	含义/结果
<code>sort.Float64s(fs)</code>	将 <code>[]float64</code> 按升序排序
<code>sort.Float64AreSorted(fs)</code>	如果 <code>[]float64</code> 是有序的则返回 <code>true</code>
<code>sort.Ints(is)</code>	将 <code>[]int</code> 按升序排序
<code>sort.IntsAreSorted(is)</code>	如果 <code>[]int</code> 是有序的则返回 <code>true</code>
<code>sort.IsSorted(d)</code>	如果 <code>sort.Interface</code> 的值 <code>d</code> 是有序的，则返回 <code>true</code>
<code>sort.Search(size, fn)</code>	在一个排序好的数组中根据函数签名为 <code>func(int)bool</code> 的函数 <code>fn</code> 进行搜索，返回第一个使得函数 <code>fn</code> 返回值为 <code>true</code> 的索引
<code>sort.SearchFloat64s(fs, f)</code>	返回有序 <code>[]float64</code> 切片 <code>fs</code> 中类型为 <code>float64</code> 的值 <code>f</code> 的索引
<code>sort.SearchInts(is, i)</code>	返回有序 <code>[]int</code> 切片 <code>is</code> 中类型为 <code>int</code> 的值 <code>i</code> 的索引
<code>sort.SearchStrings(ss, s)</code>	返回有序 <code>[]string</code> 切片 <code>ss</code> 中类型为 <code>string</code> 的值 <code>s</code> 的索引
<code>sort.Sort(d)</code>	排序类型为 <code>sort.Interface</code> 的切片 <code>d</code>
<code>sort.Strings(ss)</code>	按升序排序 <code>[]string</code> 类型的切片 <code>ss</code>
<code>sort.StringsAreSorted(ss)</code>	如果 <code>[]string</code> 类型的切片 <code>ss</code> 是有序的，返回 <code>true</code>

正如我们在之前章节中所看到的，Go语言对数值的排序方式并不奇怪。然而，字符串的排序完全是字节排序，这点我们在前面章节中已讨论过（参见3.2节）。这也意味着字符串的排序是区分大小写的。这里给出了一些字符串排序例子以及它们输出的结果。

```
files := []string{ " Test.conf " , " util.go " , " Makefile " , " misc.go " , " main.go " }
fmt.Printf( " Unsorted:           %q\n " , files)
sort.Strings(files)                // 标准库中的排序函数
fmt.Printf( " Underlying bytes: %q\n " , files)
SortFoldedStrings(files)           // 自定义排序函数
fmt.Printf( " Case insensitive: %q\n " , files)

Unsorted:           [ " Test.conf "  " util.go "  " Makefile "  "
misc.go "  " main.go " ]
Underlying bytes:[ " Makefile "  " Test.conf "  " main.go "  "
misc.go "  " util.go " ]
Case insensitive:[ " main.go "  " Makefile "  " misc.go "  " Test.conf
"  " util.go " ]
```

标准库中的`sort.Strings()`函数接受一个`[]string`切片，并将该字符串按照它们底层的字节码在原地以升序排序。如果字符串使用的是同样的字符到字节映射的编码方案（例如，它们都是在本程序中创建的，或者是由其他Go程序创建的），该函数会按码点排序。自定义的函数`SortFoldedStrings()`功能与此类似，不同的是它使用`sort`包中通用的`sort.Sort()`函数来对字符串进行大小写无关的排序。

`sort.Sort()`函数能够对任意类型进行排序，只要其类型提供了`sort.Interface`接口中定义的方法，即只要这些类型根据相应的签名实现了`Len()`、`Less()`和`Swap()`等方法。我们创建了一个自定义类型

FoldedStrings，提供了这些方法。下面是SortFoldedStrings()函数、FoldedString类型以及相应方法的实现。

```
func SortFoldedStrings(slice []string) {
    sort.Sort(FoldedStrings(slice))
}

type FoldedStrings []string

func (slice FoldedStrings) Len() int { return len(slice) }

func (slice FoldedStrings) Less(i, j int) bool {
    return strings.ToLower(slice[i]) < strings.ToLower(slice[j])
}

func (slice FoldedStrings) Swap(i, j int) {
    slice[i], slice[j] = slice[j], slice[i]
}
```

SortFoldedStrings()函数简单地使用标准库中的sort.Sort()函数来完成工作，即使用Go语言的标准转换语法将给定的[]string类型的值转换成FoldedStrings类型的值。通常，当我们基于一个内置类型创建自定义的类型时，我们可以通过这样的类型转换方法将内置的类型转换提升为自定义类型（自定义类型相关的内容将在第6章讲解）。

FoldedStrings类型实现了3个方法以对应sort.Interface接口。这几个方法都比较小巧，Less()方法通过使用strings.ToLower()函数来达到大小写无关（如果我们要以逆序排序，可以简单地将Less()方法中的小于操作符 < 改成大于操作符 >）。

正如我们在前面章节（参见3.2节）中所讨论的一样，对于7位ASCII编码（英语）的字符串而言，SortFoldedStrings()函数的实现非常完美，但是对于其他非英语语言来说却不一定能够得到完美的排序结果。对Unicode编码的字符串进行排序不是个简单的任务，相关的详

细描述在 Unicode 排序算法的文档中有解释 (unicode.org/reports/tr10)。

```
files := []string{ " Test.conf " , " util.go " , " Makefile " , " misc.go " , " main.go " }
target := " Makefile "
for i, file := range files {
    if file == target {
        fmt.Printf( " found \ " %s\ " at files[%d]\n " , file, i)
        break
    }
}
```

```
found " Makefile " at files[2]
```

对于无序数据来说，使用这样的简单的线性搜索是唯一的选择，而这对于小切片（大至上百个元素）来说效果也不错。但是对于大切片特别是如果我们需要进行重复搜索的话，线性搜索就会非常低效，平均每次都需要让一半的元素相互比较。

Go提供了一个使用二分搜索算法的`sort.Search()`方法：每次只需比较 $\log_2 n$ 个元素（其中 n 为切片中的元素总数）。从这个角度看，一个含1 000 000个元素的切片线性搜索平均需要500 000次比较，最坏时需要1 000 000次比较。而二分搜索即便是在最坏的情况下最多也只需要20次比较。

```
sort.Strings(files)
fmt.Printf( " %q\n " , files)
i := sort.Search(len(files),
    func(i int) bool {return files[i] >= target })
```

```

if i < len(files) && files[i] == target {
    fmt.Printf( " found \ " %s\ " at files[%d]\n " , files[i], i)
}
[ " Makefile " " Test.conf " " main.go " " misc.go " " util.go " ]
found " Makefile " at files[0]

```

`sort.Search()`函数接受两个参数：所处理的切片的长度和一个将目标元素与有序切片的元素相比较的函数，如果该有序切片是升序排序的则使用 `>=` 操作符，如果逆序排序则使用 `<=`操作符。该函数必须是一个闭包，即它必须创建于该切片的作用域内。因为它必须将切片当成是其自身状态的一部分。（闭包将在5.6.3节中讲解。）`sort.Search()`函数返回一个`int`型的值。只有当该值小于切片的长度并且在该索引位置的元素与目标元素相匹配时，我们才能够确定找到了需要找的元素。

以下是这个函数的一个变形，它从一个不区分大小写的有序`[]string`切片中搜索一个小写的目标字符串。

```

target := " makefile "
SortFoldedStrings(files)
fmt.Printf( " %q\n " , files)
caseInsensitiveCampare := func(i int) bool {
    return strings.ToLower(files[i]) >= target
}
i := sort.Search(len(files), caseInsensitiveCampare)
if i <= len(files) && strings.EqualFold(files[i], target) {
    fmt.Printf( " found \ " %s\ " at files[%d]\n " , files[i], i)
}
[ " main.go " " Makefile " " misc.go " " Test.conf " " util.go " ]
found " Makefile " at files[1]

```

这里我们除了调用`sort.Search()`函数之外创建了一个比较函数。注意，与前面的例子一样，该比较函数也必须是创建于切片作用域范围内的闭包。我们本可以使用代码 `strings.ToLower(files[i]) == target` 来做比较，但是这里我们使用了更为方便的`strings.EqualFold()`函数来不区分大小写地比较字符串。

Go语言的切片是一种非常强大、方便，且功能非常方便的数据结构，难以想像会有哪个不太小的Go程序不需要用到它。我们将在本章后面看到实际的使用案例（参见4.4节）。

虽然切片足以满足大多数数据结构的使用案例，但有些情况下我们不得不将数据保存为“键值”对来按键进行快速查找。这个功能由Go语言中的映射类型提供，也是本书下一节的主题。

4.3 映射

Go语言中的映射（`map`）是一种内置的数据结构，保存键-值对的无序集合，它的容量只受到机器内存的限制 [8]。在一个映射里所有的键都是唯一的而且必须是支持`==`和`!=`操作符的类型，大部分Go语言的基本类型都可以作为映射的键，例如，`int`、`float64`、`rune`、`string`、可比较的数组和结构体、基于这些类型的自定义类型，以及指针。Go语言的切片和不能用于比较的数组和结构体（这些类型的成员或者字段不支持`==`或者`!=`操作）或者基于这些的自定义类型则不能作为键。指针、引用类型或者任何内置类型的值、自定义类型都可以用做值，包括映射本身，所以它可以创建任意复杂的数据结构。表4-3列出了Go语言中映射支持的操作。

表4-3 映射的操作

语法	含义/结果
<code>m[k] = v</code>	用键 <code>k</code> 来将值 <code>v</code> 赋值给映射 <code>m</code> 。如果映射 <code>m</code> 中的 <code>k</code> 已存在，则将之前的值丢弃
<code>Delete(m, k)</code>	将键 <code>k</code> 及其相关的值从映射 <code>m</code> 中删除。如果 <code>k</code> 不存在则安全地不执行任何操作
<code>v := m[k]</code>	从映射 <code>m</code> 中取得键 <code>k</code> 相对应的值并将其赋值给 <code>v</code> 。如果 <code>k</code> 在映射中不存在，则将映射类型的 0 值赋值给 <code>v</code>
<code>v, found := m[k]</code>	从映射 <code>m</code> 中取得键 <code>k</code> 相对应的值并将其赋值给 <code>v</code> ，并将 <code>found</code> 的值赋值为 <code>true</code> 。如果 <code>k</code> 在映射中不存在，则将映射类型的 0 值赋值给 <code>v</code> ，并将 <code>found</code> 的值赋值为 <code>false</code>
<code>len(m)</code>	返回映射 <code>m</code> 中的项（“键/值”对）的数目

因为映射属于引用类型，所以不管一个映射保存了多少数据，传递都是很廉价的（在64 位机器上只需要8个字节，在32位机器上只需要4字节）。映射的查询很快，甚至比线性搜索还快。从非正式的实验结果来看 [9]，尽管映射的查找比数组或者切片里的直接索引慢两个数量级左右（即100倍），但在需要用到映射的地方来说，这仍然是非常快的了，实际使用中几乎不大可能出现性能问题。图4-5展示了一个 `map[string]float64` 类型的映射示意图。

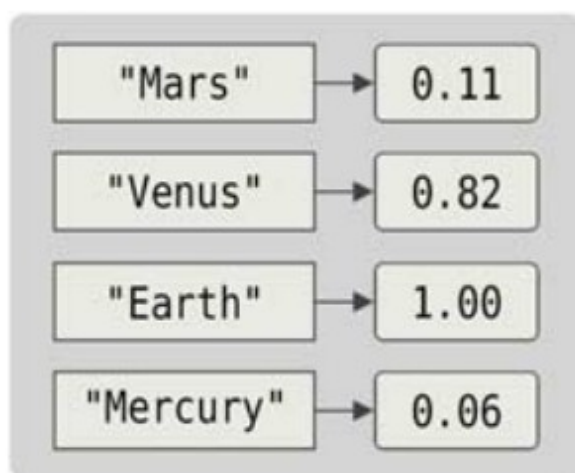


图4-5 一个键为string类型、值为float64类型的映射的剖析

由于 `[]byte` 是一个切片，不能作为映射的键，但是我们可以先将 `[]byte` 转换成字符串，例如 `string([]byte)`，然后作为映射的键字段，等有需要的时候再转换回来，这种转换并不会改变原有切片的数据。

映射里所有键的数据类型必须是相同的，值也必须如此，但键和值的数据类型可以不同（通常都不相同）。但是如果值的类型是接口类型，我们就可以将一个满足这个接口定义的值作为映射的值，甚至我们可以创建一个值为空接口（`interface{}`）的映射，这就意味着任意类型的值都可以作为这个映射的值。不过当我们需要访问这个值的时候，需要使用类型开关和类型断言获得这个接口类型的实际类型，或者也可以通过类型检视（`type introspection`）来获得变量的实际类型。

（接口会在第6章介绍，反射会在第9章介绍。）

映射的创建方式如下：

```
make(map[KeyType]ValueType, initialCapacity)
```

```
make(map[KeyType]ValueType)
```

```
map[KeyType]ValueType{}
```

```
map[KeyType]ValueType{key1: value1, key2: value2,..., keyN: valueN}
```

Go语言内置的`make()`函数可用来创建切片、映射和通道。当用`make()`来创建一个映射时，实际上得到一个空的映射，如果指定容量（`initialCapacity`）就会预先申请到足够的内存，并且随着加入的项越来越多，映射会自动扩容。第二种写法和第三种写法是完全一样的，最后两种写法使用的是复合语法，实际编程中这是非常方便的，比如创建一个空的映射或者具有某些初始值的映射。

4.3.1 创建和填充映射

我们来创建一个映射结构，并往里面填充一些数据，键为`string`类型，值是`float64`类型。

```
massForPlanet := make(map[string]float64) // 与 map[string]float64{}  
相同
```



```
massForPlanet[ " Mercury " ] = 0.06
massForPlanet[ " Venus " ] = 0.82
massForPlanet[ " Earth " ] = 1.00
massForPlanet[ " Mars " ] = 0.11
fmt.Println(massForPlanet)
```

```
map[Venus:0.82 Mars:0.11 Earth:1 Mercury:0.06]
```

对于一些比较小的映射，我们没必要考虑是否要指定它们的初始容量，但如果这个映射比较大，指定恰当的容量可以提高性能。通常如果你知道它的容量的话最好就指定它，即使是近似的也好。

映射和数组或切片一样可以使用索引操作符[]，但和数组或切片不同的是，映射的键类型不必是int型的，例如我们现在用的是string类型的键。

我们使用了 `fmt.Println()` 函数打印映射，这个函数使用 `%v` 格式符将映射中每项内容都以“键：值”的形式打印出，并且项与项之间以空格分开，因为映射里面的项都是无序的，所以在其他机器上打印出来的结果顺序可能和本书的不一样。

之前提过，映射的键可以是一个指针，我们下面将会看到一个例子，它的键的类型是 `*Point`，`Point` 定义如下：

```
type Point struct{ x, y, z int }
func (point Point) String() string {
    return fmt.Sprintf( " (%d,%d,%d) " , point.x, point.y, point.z)
}
```

`Point` 类型里有了3个int类型的变量，还有一个 `String()` 的方法，这样就确保了当我们打印一个 `*Point` 时Go语言会调用它的 `String()` 方法而不是简单地输出 `Point` 的内存地址。

顺便说一句，我们可以使用%p格式符来强制Go语言打印出内存地址，格式符此前介绍过（参见3.5.6节）。

```
triangle := make(map[*Point]string, 3)
triangle[&Point{89, 47, 27}] = " α "
triangle[&Point{86, 65, 86}] = " β "
triangle[&Point{7, 44, 45}] = " γ "
fmt.Println(triangle)
```

```
map[(7,44,45):γ (89,47,27):α (86,65,86):β]
```

这里我们创建了一个初始存储大小为3的映射，然后往里添加指针类型的键和字符串值。使用复合语法创建每个 **Point** 值，然后用**&**操作符取得指针作为键，这样键就是一个***Point**指针而不是一个**Point**类型的值。因为**Point**实现了**String()**方法，所以打印映射的时候我们能以可读的方式看到 ***Point**的值。

使用指针作为映射的键意味着我们可以增加两个相同的内容，只要分别创建它们就可以获得不同的地址。但如果我们希望这个映射对任何实际上相同的内容只存储一个的话会怎么样呢？这也是很容易的，只要我们存储**Point**的值而不是指向**Point**的指针即可，要知道，Go语言允许将一个结构体作为映射的键，只要它们所有的字段都支持**==**和**!=**运算即可。下面是一个例子。

```
nameForPoint := make(map[Point]string) // 等 同 于 :
map[Point]string{}
nameForPoint[Point{54, 91, 78}] = " x "
nameForPoint[Point{54, 158, 89}] = " y "
fmt.Println(nameForPoint)
```

```
map[(54,91,78):x (54,158,89):y]
```

nameForPoint的每一个键都是唯一的Point结构，我们可以在任何时候改变它所映射的字符串。

```
populationForCity := map[string]int{ " Istanbul " : 12610000,
    " Karachi " : 10620000, " Mumbai " : 12690000, " Shanghai " :
13680000}

for city, population := range populationForCity {
    fmt.Printf( " %-10s %8d\n " , city, population)
}
```

Shanghai	13680000
Mumbai	12690000
Istanbul	12610000
Karachi	10620000

这是我们这一节最后的例子了，同样，我们使用复合语法创建了整个映射结构。当用一个for...range 循环来遍历映射的时候，对于映射中的每一项都返回两个变量键和值，直到遍历完所有的键/值对或者循环被打破。如果只关心其中一个变量的话，因为映射里的项都是无序的，我们并不知道每次迭代实际返回的是哪一个，更多的时候我们只是获得所有遍历出来的项并更新它们，所以遍历出来的次序不重要。但是，如果我们想按照某种方式来遍历，如按键序，这也不难，很快我们就可以看到了（见4.3.4节）。

4.3.2 映射查询

Go语言提供了两种类似的语法用于映射查询，两种方式都是使用[]操作符。下面是一种最简单的方法。

```
population := populationForCity[ " Mumbai " ]
fmt.Println( " Mumbai's population is " , population)
```

```
population = populationForCity[ " Emerald City " ]  
fmt.Println( " Emerald City's population is " , population)
```

```
Mumbai's population is 12690000
```

```
Emerald City's population is 0
```

如果我们查询的键出现在映射里面，那就返回它对应的值，如果这个键并没有在映射里，就会返回一个 0 值，但是 0 值也有可能是因为这个键不存在，所以这里我们不能简单地认为“Emerald City”这个城市的人口数就是0，或者说这个城市并不在这个映射中。Go语言还有一种语法解决了这个问题。

```
city := " Istanbul "  
if population, found := populationForCity[city]; found {  
    fmt.Printf( " %s's population is %d\n " , city, population)  
} else {  
    fmt.Printf( " %s's population data is unavailable\n " , city)  
}  
city = " Emerald City "  
_, present := populationForCity[city]  
fmt.Printf( " %q is in the map == %t\n " , city, present)
```

```
Istanbul's population is 12610000
```

```
" Emerald City " is in the map == false
```

当我们使用索引操作符[]来查找映射中的键的时候，我们指定两个返回变量，第一个用来获得键对应的值（如果键不存在的话会返回0值），第二个变量是一个布尔类型（键存在则为true，否则为false），这样我们就可以知道这个键是否真的在映射里。像上述代码里面的第二个查询一样，如果我们只是关心某个键是否存在的话可以使用空变量（一个下划线）来接受值。

4.3.3 修改映射

我们可以往映射里插入或者删除一项，所谓项（item），也就是一个“键/值”对，任何一项的值都可以修改，如下。

```
fmt.Println(len(populationForCity), populationForCity)
delete(populationForCity, " Shanghai " )    // 删除
fmt.Println(len(populationForCity), populationForCity)
populationForCity[ " Karachi " ] = 11620000 // 更新
fmt.Println(len(populationForCity), populationForCity)
populationForCity[ " Beijing " ] = 11290000 // 插入
fmt.Println(len(populationForCity), populationForCity)

4  map[Shanghai:13680000 Mumbai:12690000 Istanbul:12610000
Karachi:10620000]
3 map[Mumbai:12690000 Istanbul:12610000 Karachi:10620000]
3 map[Mumbai:12690000 Istanbul:12610000 Karachi:11620000]
4  map[Mumbai:12690000 Istanbul:12610000 Karachi:11620000
Beijing:11290000]
```

插入和更新一个项的语法是完全一样的，如果给定一个键对应的项不存在，那么映射默认会创建一个新的项来保存这个“键/值”对，否则，就将这个键原来的值设置成新的值。如果我们尝试去删除映射里一个不存在的项，Go语言并不会做任何不安全的事情。

对于键不能用同样的方式修改，但可以用如下这种效果相同的做法：

```
oldKey, newKey := " Beijing " , " Tokyo "
value := populationForCity[oldKey]
delete(populationForCity, oldKey)
populationForCity[newKey] = value
```

```
fmt.Println(len(populationForCity), populationForCity)
```

```
4 map[Mumbai:12690000 Istanbul:12610000 Karachi:11620000  
Tokyo:11290000]
```

我们先得到键的值，然后删除这个键对应的项，接着创建一个新的项用来保存新的键和原来的值。

4.3.4 键序遍历映射

当使用数据时，我们通常需要按照某种被认可的方式将这些数据显示出来，下面的例子展示了如何按字典序（严格来说，是Unicode码点的顺序）显示populationForCity里的城市数据。

```
cities := make([]string, 0, len(populationForCity))  
for city := range populationForCity {  
    cities = append(cities, city)  
}  
sort.Strings(cities)  
for _, city := range cities {  
    fmt.Printf( " %-10s %8d\n ", city, populationForCity[city])  
}
```

```
Beijing 11290000  
Istanbul 12610000  
Karachi 11620000  
Mumbai 12690000
```

首先我们创建一个类型为[]string的切片，Go语言会默认将其初始化为0值，但设置了足够大的容量去保存映射里的键。然后将我们遍历映射得到的所有键（因为我们现在只会用到一个变量city，所以不需要

得到完整的一个“键/值”对）追加到这个切片 `cities` 里去。下一步，对 `cities` 排序，再遍历 `cities`（使用空变量忽略 `int` 型的索引），查询每个 `city` 对应的城市人口数量。

这个算法的思想是，创建一个足够大的切片去保存映射里所有的键，然后对切片排序，遍历切片得到键，再从映射里得到这个键的值，这样就可以实现顺序输出了。一般希望按键序来遍历一个映射都可以这样做。

另一种方法就是使用有序的数据结构，例如一个有序映射，我们在后面的章节会介绍一个例子（6.5.3节）。

按值排序也是可以的，例如将一个映射反转，下一节将提到这个方法。

4.3.5 映射反转

如果一个映射的值都是唯一的，且值的类型也是映射所支持的键类型的话，我们就可以很容易地将它反转。

```
cityForPopulation := make(map[int]string, len(populationForCity))
for city, population := range populationForCity {
    cityForPopulation[population] = city
}
fmt.Println(cityForPopulation)
```

```
map[12610000:Istanbul 11290000:Beijing 12690000:Mumbai
11620000:Karachi]
```

因为 `populationForCity` 是 `map[string]int` 类型的，所以我们创建一个 `map[int]string` 类型的 `cityForPopulation`。然后遍历 `populationForCity`，

并将得到的键和值反转，插入 `cityForPopulation` 里去，也就是说，原先的值将作为键，而原先的键则作为现在的值。

当然，如果原来映射的值不是唯一，反转就会失败，实质上只有最后一个不唯一的值对应的键被保存。可以通过创建一个多值的映射来解决这个问题，如在我们这个例子里，反转后的映射的类型是 `map[int][]string`（键是 `int` 类型的，值是 `[]string`），很快我们就可以看到一个实际的例子（参见4.4.2节）。

4.4 例子

这一节我们来看两个小例子，第一个是关于一维或者二维切片的，第二个主要是映射，包括当映射的值不唯一时如何反转，也涉及切片和排序。

4.4.1 猜测分隔符

有时候我们可能收到很多数据文件需要去处理，每个文件每一行就是一条记录，但是不同的文件可能使用不同的分隔符（例如，可能是制表符、空白符或者“*”等）。为了能够大批量地处理这些文件我们必须能够判断每个文件所用的分隔符，这一节展示的 `guess_separator` 例子（在文件 `guess_separator/guess_separator.go` 里）尝试去判断所有给定文件的分隔符。

运行示例如下：

```
./guess_separator information.dat  
tab-separated
```


程序从给定文件里读取前 5 行（如果文件行数小于 5 则全读取进来），然后分析所用的分隔符。我们从main()函数以及它所调用的函数开始分析（除去例程），import部分略过。

```
func main() {
    if len(os.Args) == 1 || os.Args[1] == "-h" || os.Args[1] == "--help" {
        fmt.Printf("usage: %s file\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    separators := []string{"\t", "*", "|", "."}
    linesRead, lines := readUpToNLines(os.Args[1], 5)
    counts := createCounts(lines, separators, linesRead)
    separator := guessSep(counts, separators, linesRead)
    report(separator)
}
```

main()函数首先检查命令行是否指定了文件，如果一个都没有，就打印一条帮助消息然后退出程序。我们创建了一个[]string切片来保存我们感兴趣的分隔符列表。按照惯例，对于使用空白分隔符的文件，我们当成是“”来处理（空字符串）。

第一步数据处理是从文件中读取前5行内容。这里没有显示readUpNLines()函数的代码，因为我们之前就有几个这样从文件中读取行的例子。和之前例子不同的是，readUpToNLines()函数只是读取一定数量的行，如果文件实际的行数比指定的小，那就全读，最后返回实际读取了的行数及每行的数据。

然后就是createCounts()函数，代码如下。

```
func createCounts(lines, separators []string, linesRead int) [][]int {
    counts := make([][]int, len(separators))
```

```

    for sepIndex := range separators {
        counts[sepIndex] = make([]int, linesRead)
        for lineIndex, line := range lines {
            counts[sepIndex][lineIndex] =
                strings.Count(line, separators[sepIndex])
        }
    }
    return counts
}

```

`createCounts()`的目的就是计算出一个保存了每一个分隔符在每行出现的次数的矩阵。

函数首先创建一个`[][]int`类型的二维切片 `counts`，大小和`main` 函数里的`separators`一样。如果有4个分隔符，那么`counts`的值是`[nil nil nil nil]`，外围的`for`循环将每一个`nil`替换成`[]int`，用来保存每个分隔符在每一行出现的次数，所以每一个`nil`都被替换成了`[0 0 0 0 0]`，注意Go语言默认总是将一个值初始化为0值。

内部的`for`循环是用来计算`counts`矩阵的。每一行里每一个分隔符出现的次数都会被统计下来并相应地更新 `counts`的值，`strings.Count()`函数返回它的第二个参数指定的字符串在第一个参数指定的字符串里出现的次数。

例如，对于一个使用制表符分隔的文件，其中某些字段包含圆点符号、空格和星号，我们可能得到这样一个`counts`矩阵：`[[3 3 3 3 3] [0 0 4 3 0] [0 0 0 0] [1 2 2 0 0]]`。`counts`里的每一项都是`[]int`类型的切片，保存了每一个分隔符（制表符、星号、竖杠、圆点）在每一行里出现的次数。从这个数字看来每一行里都出现了3个制表符，有两行出现星号（一行3个，一行4个），有3行出现圆点，没有一行出现竖杠。对我们

来说很明显制表符是分隔符，当然程序必须得自己发现这个，它是用 `guessSep()` 函数来完成这个功能的。

```
func guessSep(counts [][]int, separators []string, linesRead int) string {
    for sepIndex := range separators {
        same := true
        count := counts[sepIndex][0]
        for lineIndex := 1; lineIndex < linesRead; lineIndex++ {
            if counts[sepIndex][lineIndex] != count {
                same = false
                break
            }
        }
        if count > 0 && same {
            return separators[sepIndex]
        }
    }
    return " "
```

`guessSep()`是这样处理的：如果某个分隔符在每一行出现的次数都是相同的（但不能是0值），就认为文件使用的就是这个分隔符。外部的for循环检查每一个分隔符，内部for循环检查分隔符在每行出现的次数。`same`变量初始化为`true`，默认是假设当前分隔符在每行出现的次数都是一样的，`count`为当前分隔符在第一行出现的次数。然后内循环开始，如果发现有一行的次数和`count`不一样，`same`的值变为`false`，内循环退出，然后尝试下一个分隔符。如果内循环没有将`false`赋值给`same`变量，而且`count`的值大于0，就表示已经找到我们想要的分隔符了，并

立即返回它。最后如果没有找到分隔符，返回一个空的字符串，也就是说所有的行都是以空白符分隔的，或者完全没有分隔。

```
func report(separator string) {  
    switch separator {  
    case " " :  
        fmt.Println( " whitespace-separated or not separated at all " )  
    case "\t" :  
        fmt.Println( " tab-separated " )  
    default:  
        fmt.Printf( " %s-separated\n " , separator)  
    }  
}
```

`report()`这个函数不怎么重要，只是显示文件所用的分隔符是什么。

我们从这个例子了解到两种切片的典型用法，一维的和二维的（`separators`、`lines`，还有`counts`），下一个例子我们将会看到映射、切片还有排序。

4.4.2 词频统计

文本分析的应用很广泛，从数据挖掘到语言学习本身。这一节我们来分析一个例子，它是文本分析最基本的一种形式：统计出一个文件里单词出现的频度。

频度统计后的结果可以以两种不同的方式显示，一种是将单词按照字母顺序把单词和频度排列出来，另一种是将频度按照有序列表的方式把频度和对应的单词显示出来。`wordfrequency` 程序（在文件 `wordfrequency/wordfrequency.go`里）生成两种输出，如下所示。

```
$/wordfrequency small-file.txt
```

Word	Frequency
------	-----------

ability	1
---------	---

about	1
-------	---

above	3
-------	---

...

years	1
-------	---

you	128
-----	-----

Frequency → Words

1	ability, about, absence, absolute, absolutely, abuse, accessible,...
---	--

2	accept, acquired, after, against, applies, arrange, assumptions,...
---	---

...

128	you
-----	-----

151	or
-----	----

192	to
-----	----

221	of
-----	----

345	the
-----	-----

即使是很小的文件，单词的数量和不同频度的数量都可能会非常大，篇幅有限，我们只显示部分结果。

第一种输出是比较直接的，我们可以使用一个`map[string]int`类型的结构来保存每一个单词的频度。但是要得到第二种输出结果我们需要将整个映射进行反转，但这并不是那么容易，因为很可能具有相同的频度的单词不止一个，解决的方法就是反转成多值类型的映射，如`map[int][]string`，也就是说，键是频度而值则是所有具有这个频度的单词。

我们将从程序的`main()`函数开始，从上到下分析，和通常一样，忽略掉`import`部分。

```

func main() {
    if len(os.Args) == 1 || os.Args[1] == "-h" || os.Args[1] == "--help" {
        fmt.Printf(" usage: %s <file1> [<file2> [...<fileN>]]\n",
            filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    frequencyForWord := map[string]int{} // 与 make(map[string]int) 相同
    for _, filename := range commandLineFiles(os.Args[1:]) {
        updateFrequencies(filename, frequencyForWord)
    }
    reportByWords(frequencyForWord)
    wordsForFrequency := invertStringIntMap(frequencyForWord)
    reportByFrequency(wordsForFrequency)
}

```

`main()`函数首先分析命令行参数，之后再进行相应处理。

我们使用复合语法创建一个空的映射，用来保存从文件读到的每一个单词和对应的频度。接着我们遍历从命令行得到的每一个文件，分析每一个文件后更新`frequencyForWord`的数据。

得到第一个映射之后，我们就输出第一个报告：一个按照字母表顺序排序的单词列表和对应的出现频率。然后我们创建一个反转的映射，输出第二个报告：一个排序的出现频率列表和对应的单词。

```

func commandLineFiles(files []string) []string {
    if runtime.GOOS == "windows" {
        args := make([]string, 0, len(files))
        for _, name := range files {

```

```

        if matches, err := filepath.Glob(name); err != nil {
            args = append(args, name) // 无效模式
        } else if matches != nil { // 至少有一个匹配
            args = append(args, matches...)
        }
    }
    return args
}
return files
}

```

因为在Unix类系统（如Linux或Mac OS X等）的shell默认会自动处理通配符（也就是说，*.txt能匹配任意后缀为.txt的文件，如README.txt和INSTALL.txt等），而Windows平台的shell程序（cmd.exe）不支持通配符，所以如果用户在命令行输入，如*.txt，那么程序只能接收到*.txt。为了保持平台之间的一致性，我们使用commandLineFiles()函数来实现跨平台的处理，当程序运行在Windows平台时，我们自己把文件名通配功能给实现了。（另一种跨平台的办法就是不同的平台使用不同的.go文件，这在9.1.1.1节有描述。）

```

func    updateFrequencies(filename    string,    frequencyForWord
map[string]int) {
    var file *os.File
    var err error
    if file, err = os.Open(filename); err != nil {
        log.Println( " failed to open the file: " , err)
        return
    }
    defer file.Close()
}

```

```

        readAndUpdateFrequencies(bufio.NewReader(file),
frequencyForWord)
    }

```

updateFrequencies() 函数纯粹就是用来处理文件的。它打开给定的文件，并使用defer让函数返回时关闭文件句柄。这里我们将文件作为一个 *bufio.Reader （使用 bufio.NewReader() 函数创建）传给 readAndUpdateFrequencies()函数，因为这个函数是以字符串的形式一行一行地读取数据的而不是读取字节流。可见，实际的工作都是在 readAndUpdateFrequencies()函数里完成的，代码如下。

```

func      readAndUpdateFrequencies(reader      *bufio.Reader,
frequencyForWord map[string]int) {
    for {
        line, err := reader.ReadString('\n')
        for _, word := range SplitOnNonLetters(strings.TrimSpace(line)) {
            if len(word) > utf8.UTFMax || utf8.RuneCountInString(word) >
1 {
                frequencyForWord[strings.ToLower(word)] += 1
            }
        }
        if err != nil {
            if err != io.EOF {
                log.Println( " failed to finish reading the file: " , err)
            }
            break
        }
    }
}

```


第一部分的代码我们应该很熟悉了。我们用了一个无限循环来一行一行地读一个文件，当读到文件结尾或者出现错误（这种情况下我们将错误报告给用户）的时候就退出循环，但我们并不退出程序，因为还有很多其他的文件需要去处理，我们希望做尽可能多的工作和报告我们捕获到的任何问题，而不是将工作结束在第一个错误上面。

内循环就是处理结束的地方，也是我们最感兴趣的。任意一行都可能包括标点、数字、符号或者其他非单词字符，所以我们逐个单词地去读，将每一行分隔成单词并使用`SplitOnNonLetters()`函数忽略掉非单词的字符。而且我们一开始就过滤掉字符串开头和结束处的空白。

如果我们只关心至少两个字母的单词，最简单的办法就是使用只有一条语句的if 语句，也就是说，如果`utf8.RuneCountInString(word) > 1`，那这就是我们想要的。

刚才描述的那个简单的if 语句可能有一点性能损耗，因为它会分析整个单词。所以在这个程序里我们用了一个两个分句的if 语句，第一个分句用了一个非常高效的方法，它检查这个单词的字节数是否大于`utf8.UTFMax`（它是一个常量，值为4，用来表示一个UTF-8字符最多需要几个字节）。这是最快的测试方法，因为Go语言的strings知道它们包含了多少个字节，还有Go语言的二进制布尔操作符号总是走捷径的（2.2节）。当然，由4个或者更少字节组成的单词（例如7位的ASCII码字符或者一对2个字节的UTF-8字符）在第一次检查时可能失败，不过这不是问题，还有第二次检查（`rune`的个数）也很快，因为它通常只有4个或者不到4个字符需要去统计。在我们这个情况里需要用到两个分句的if语句吗？这就取决于输入了，越多的字符需要的处理时间就越长，就有更多可能优化的地方。唯一可以明确知道的办法就是使用真实或者典型的数据集来做基准测试。

```
func SplitOnNonLetters(s string) []string {  
    notALetter := func(char rune) bool { return !unicode.IsLetter(char) }
```

```

    return strings.FieldsFunc(s, notALetter)
}

```

这个函数在非单词字符上对一个字符串进行切分。首先我们为 `strings.FieldsFunc()` 函数创建一个匿名函数 `notALetter`，如果传入的是字符那就返回 `false`，否则返回 `true`。然后我们返回调用函数 `strings.FieldsFunc()` 的结果，调用的时候将给定的字符串和 `notALetter` 作为它的参数。（我们在之前在 3.6.1 节里讨论过 `strings.FieldsFunc()` 函数。）

```

func reportByWords(frequencyForWord map[string]int) {
    words := make([]string, 0, len(frequencyForWord))
    wordWidth, frequencyWidth := 0, 0
    for word, frequency := range frequencyForWord {
        words = append(words, word)
        if width := utf8.RuneCountInString(word); width > wordWidth {
            wordWidth = width
        }
        if width := len(fmt.Sprint(frequency)); width > frequencyWidth {
            frequencyWidth = width
        }
    }
    sort.Strings(words)
    gap := wordWidth + frequencyWidth - len( " Word " ) - len( "
Frequency " )
    fmt.Printf( " Word %*s%s\n " , gap, " " , " Frequency " )
    for _, word := range words {
        fmt.Printf( " %-*s %*d\n " , wordWidth, word, frequencyWidth,
            frequencyForWord[word])
    }
}

```

```
}  
}
```

一旦计算出了 `frequencyForWord`，就调用 `reportByWords()` 将它的数据打印出来。因为我们希望输出结果是按照字母顺序排列的（实际上是按照Unicode码点顺序），所以我们首先创建一个空的容量足够大的 `[]string` 切片来保存所有在 `frequencyForWord` 里的单词，同样我们希望能知道最长的单词和最高的频度的字符宽度（比如说，频度有多少个数字），所以我们可以以整齐的方式输出我们的结果，用 `wordWidth` 和 `frequencyWidth` 来记录这两个结果。

第一个循环遍历映射里的所有项，每个单词追加到 `words` 字符串切片里去，这个操作的效率是很高的。因为 `words` 的容量足够大了，所以 `append()` 函数需要做的只是把给定的单词追加到第 `len(words)` 个索引位置上去，`words` 的长度会自动增加1。

对于每一个单词我们统计它包含的字符的数量，如果这个值比 `wordWidth` 大就将它设置为 `wordWidth` 的值。同样地，我们统计表示一个频度所需要的字符数。我们可以安全地使用 `len()` 函数来统计字节数，因为 `fmt.Sprint()` 函数需要传入一个数字然后返回一个全部都是7位ASCII码的字符串。这样第一个循环结束了，我们就得到了我们想要的两列数据。

得到了 `words` 切片之后，我们对它进行排序，我们不必担心是否区分大小写，因为所有的单词都是小写的，这个在 `readAndUpdateFrequencies()` 函数中已经处理好了。

经过排序之后我们打印两列标题，第一个是“Word”，为了能让 `Frequency` 最后一个字符y右对齐，我们在“Word”后打印一些空格，这是通过 `%*s` 格式化动作来实现的打印固定长度的空白。另一种办法是可以使用 `%s` 来打印 `strings.Repeat(" ", gap)` 返回的字符串。（我们之前的3.5节里讲过字符串格式化。）

最后，我们将单词和它们的频度用两列方式按照字母顺序打印出来。

```
func invertStringIntMap(intForString map[string]int) map[int][]string {
    stringsForInt := make(map[int][]string, len(intForString))
    for key, value := range intForString {
        stringsForInt[value] = append(stringsForInt[value], key)
    }
    return stringsForInt
}
```

上面的函数首先创建一个空的映射，用来保存反转的结果。但是我们不知道到底它将要保存多少个项，因此我们就先假定它和原来的映射容量一样大，毕竟不可能比原来的多。然后我们简单地遍历原来的映射，然后将它的值作为键保存到反转的映射里，并将键增加到对应的值里去，新的映射的值就是一个字符串切片，即使原来的映射有多个键对应同一个值，也不会丢掉任何数据。

```
func reportByFrequency(wordsForFrequency map[int][]string) {
    frequencies := make([]int, 0, len(wordsForFrequency))
    for frequency := range wordsForFrequency {
        frequencies = append(frequencies, frequency)
    }
    sort.Ints(frequencies)
    width := len(fmt.Sprint(frequencies[len(frequencies)-1]))
    fmt.Println( " Frequency → Words " )
    for _, frequency := range frequencies {
        words := wordsForFrequency[frequency]
        sort.Strings(words)
```

```

        fmt.Printf( " %*d %s\n " , width, frequency, strings.Join(words,
        " , " ))
    }
}

```

这个函数的结构和`reportByWords()`函数很相似。它首先创建一个切片用来保存频度，这个切片会按照频度升序排列。然后再计算需要容纳的频度的最大长度并以此作为第一列的宽度。之后输出报告的标题。最后，遍历输出所有的频度并按照字母升序输出对应的单词。如果一个频度有超过两个对应的单词则单词之间使用逗号分隔开。

至此我们就讲完了这章的两个完整示例，相信大家对 Go 语言指针的使用已经有了一定的了解，关键是 Go 语言中强大的切片和映射类型。在下一章中我们将讨论如何创建一个自定义函数，过程编程部分将到下章结束。之后的章节我们将接着讲解 Go 语言的面向对象编程，面向对象编程之后我们会继续讲解并发编程。

4.5 练习

本章一共有5个练习，每一个练习需要写一个小函数，以复习本章所阐述的关于切片和映射的内容。我们将5个函数放在同一个`.go`文件（`chap4_ans/chap_ans.go`）里了，同时添加了一个`main()`函数，以便更好地使用这些函数做一些简单的测试。（本书覆盖了适当的单元测试内容，详见9.1.1.3节。）

(1) 创建一个函数以接受一个`[]int`切片并返回一个`[]int`切片，其中返回的切片为传入切片的副本，只是将其中重复的内容删除了。例如，给定一个参数`[]int{9, 1, 9, 5, 4, 4, 2, 1, 5, 4, 8, 8, 4, 3, 6, 9, 5, 7, 5}`，该函数应该返回`[]int{9, 1, 5, 4, 2, 8, 3, 6, 7}`。在文件`chap4_ans.go`中，该

函数叫做UniqueInts()。该函数使用组合语法而非内置的make()函数，只有11行的长度，应该非常容易写出来。

(2) 创建一个函数接受一个[][]int切片（二维的整型切片），然后返回一个[]int切片，其中包含二维切片中的第一个切片，接着是二维切片中的第二个切片等。例如，如果该函数名为Flatten():

```
irregularMatrix := [][]int{{1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11},
    {12, 13, 14, 15},
    {16, 17, 18, 19, 20}}
slice := Flatten(irregularMatrix)
fmt.Printf( " 1x%d: %v\n " , len(slice), slice)
```

```
1x20: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]
```

该函数在文件chap4_ans.go文件中只有9行。为了让其在内部切片的长度不一致时也能正常工作（正如irregularMatrix所示），该函数做了一些额外的事情，不过其做法还是比较容易理解的。

(3) 创建一个接受[]int切片和一个列数量（整型值）参数的函数，然后返回一个[][]int切片，其所有内部切片的长度与给定的列数量参数相同。例如，如果该参数为[]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}，这里有些样本结果，每一个结果的开头都是传入的列数量:

```
3 [[1 2 3] [4 5 6] [7 8 9] [10 11 12] [13 14 15] [16 17 18] [19 20 0]]
4 [[1 2 3 4] [5 6 7 8] [9 10 11 12] [13 14 15 16] [17 18 19 20]]
5 [[1 2 3 4 5] [6 7 8 9 10] [11 12 13 14 15] [16 17 18 19 20]]
6 [[1 2 3 4 5 6] [7 8 9 10 11 12] [13 14 15 16 17 18] [19 20 0 0 0 0]]
```

需注意的是，这里有20个整型，无论是3列还是6列，都不能完全整除，因此如果需要，我们在最后一个切片的末尾填充上0，以保证所有列（即内部切片）的长度相同。

chap4_ans.go文件里的Make2D()函数一共12行，并且使用了一个7行长的辅助函数。该Make2D()函数及其帮助函数需要一定的思考才能写出来，但也不是太难。

（4）创建一个接受[]string切片参数的函数，其中该切片包含一个.ini文件格式的内容，并返回一个 map[string]map[string]类型，其键为组名，而值则为“键-值”映射组成的映射组。空行与以；开头的行需忽略。每一组在其所在的行中以一个方括号包围的名字标识，同时每一组的键和值以一行或者更多行的“键-值”的形式给出。这里有一个该函数需处理的示例[]string切片。

```
iniData := []string{
    "; Cut down copy of Mozilla application.ini file ",
    " ",
    "[App] ",
    " Vendor=Mozilla ",
    " Name=Iceweasel ",
    " Profile=mozilla/firefox ",
    " Version=3.5.16 ",
    "[Gecko] ",
    " MinVersion=1.9.1 ",
    " MaxVersion=1.9.1.* ",
    "[XRE] ",
    " EnableProfileMigrator=0 ",
    " EnableExtensionManager=1 ",
}
```

给定该数据，函数需返回以下映射，我们将其漂亮地打印出来使得其结构更容易阅读。

```
map[Gecko: map[MinVersion: 1.9.1
                MaxVersion: 1.9.1.*]
     XRE: map[EnableProfileMigrator: 0
              EnableExtensionManager: 1]
     App: map[Vendor: Mozilla
              Profile: mozilla/firefox
              Name: Iceweasel
              Version: 3.5.16]]
```

ParseIni()函数假设任何一个不在组范围内的“键-值”对都是通用的。它有 24 行长，并且可能需要花点心思才能做好。

(5) 创建一个接受一个 `map[string]map[string]string` 参数的映射，它代表一个.ini 文件中的数据。该函数需将数据以.ini 文件的形式按组字母排序输出，并且组内的键也以字母排序，同时每一行以空格分隔。例如，给定来自上一个练习中的数据，其输出应该是这样的：

```
[App]
Name=Iceweasel
Profile=mozilla/firefox
Vendor=Mozilla
Version=3.5.16
[Gecko]
MaxVersion=1.9.1.*
MinVersion=1.9.1
[XRE]
EnableExtensionManager=1
EnableProfileMigrator=0
```


PrintIni()函数一共21行，并且应该比前一个练习中的ParseIni()函数更简单。

[1].本章我们将会看到，Go语言的delete()函数用于从映射中删除键。

[2].当写作本书时，这里的成本数值是在一台32位和一台64位的机器上得到的。这个数值属于实现细节，可能随版本更新而变动，但它肯定不会变得特别大。

[3].当写作本书时，这里的成本数值是在一台32位和一台64位的机器上得到的。这个数值属于实现细节，可能随版本更新而变动，但它肯定不会变得特别大。

[4].C/C++程序员需要特别了解的是，虽然Go编译器可能在内部将栈和堆的内存区分对待，但是Go语言程序员从不需要担心这些，因为Go语言自己会在内部处理好内存管理的事情。

[5].Go语言没有也不需要C/C++中使用的->解引用操作符。(点)操作符对于绝大多数场景已经足够（比如访问一个结构体或者结构体指针内的字段），对于不满足的情况，我们可以使用多个*操作符来进行显式解引用。

[6].本书撰写时，Go语言的官方文档中通常使用术语array（数组）来描述那些实际上是切片的参数。

[7].有一个编译器是这样实现的。当一个值类型的方法被创建时，假设叫Method()，会有一个带有相同名字和签名的包装方法同时被创建，这个包装方法的接收器是一个指针，对应的就是这样的：func (value *Type) Method(){return (*value).Method()}。

[8].Go语言中的映射在某些其他场合也会被称为散列映射（hash map）、散列表（hash table）、无序映射（Unordered map）、字典或关联数组。

[9].本书撰写时还没有关于映射的时间复杂度数据。

第5章 过程式编程

本章的目的是完全覆盖自本书开始处所提及的Go过程式编程。Go语言可以用于写纯过程式程序，用于写纯面向对象程序，也可以用于写过程式和面向对象相结合的程序。学习Go语言的过程式编程至关重要，因为与Go语言的并发编程一样，面向对象编程也是建立在面向过程的基础之上的。

前面几章描述并阐明了Go语言内置的数据类型，在这一过程中，我们接触了Go语言的表达式语句和控制结构，以及许多轻量的自定义函数。本章中我们将更详细地讲解Go语言的表达式语句和控制结构，同时更加详细地讲解创建和使用自定义的函数。表5-1提供了一个Go语言的内置类型函数的列表，其中大部分已在本章内容中覆盖[\[1\]](#)。

本章有些知识点在前面的章节中已经提及过，而有些知识点涉及Go语言编程的其他方面则在接下来的章节中讲解，读者需要根据实际情况参阅前后章节的相关内容。

5.1 语句基础

形式上讲，Go语言的语法需要使用分号（`;`）来作为上下文中语句的分隔结束符。然而，如我们所见，在实际的Go程序中，很少使用到分号。那是因为编译器会自动在以标识符、数字字面量、字母字面量、字符串字面量、特定的关键字（`break`、`continue`、`fallthrough`和

`return`)、增减操作符 (`++`或者`--`) 或者是一个右括号、右方括号和右大括号 (即)、`]`、`}`) 结束的非空行的末尾自动加上分号。

有两个地方必须使用分号，即当我们需要在一行中放入一条或者多条语句时，或者是使用原始的`for`循环时 (参见5.3节)。

自动插入分号的一个重要结果是一个右大括号无法自成一 行。

// 正确代码
通过编译)

<code>for i := 0; i < 5; i++ {</code>	<code>for i := 0; i < 5; i++</code>
<code> fmt.Println(i)</code>	<code>{</code>
<code>}</code>	<code> fmt.Println(i)</code>
	<code>}</code>

上面右边的代码不能编译，因为编译器会往 `++` 后面插入一个分号。类似地，如果我们有一个无限循环 (`for`)，其左括号从下一行开始，那么编译器就会在`for`后面加上一个分号，代码同样不能编译。

表5-1 内置函数

语法	含义/结果
<code>append(s, ...)</code>	如果切片 <code>s</code> 的容量足够，则将函数末尾的项添加进给定的切片中；否则新建一个切片，其内容为原始切片的项和函数末尾传入的项（参见 4.2.3 节）
<code>cap(x)</code>	切片 <code>x</code> 的容量，或者通道 <code>x</code> 的缓存容量，或者数组 <code>x</code> （或者所指向数组）的长度。同时参考 <code>len()</code> （参见 4.2 节）
<code>close(ch)</code>	关闭通道 <code>ch</code> （但用于只接收信息的通道是非法的）。不能再往通道中发送数据。数据还可以从关闭的通道中接收（例如，任何已发送但未接收的值），并且如果通道中没有值了，接收端得到的将是该通道类型的零值
<code>complex(r, i)</code>	一个 <code>complex128</code> 复数，其实部 <code>r</code> 和虚部 <code>i</code> 给定，并且都为 <code>float64</code> （参见 2.3.2 节）
<code>copy(dst, src)</code>	将 <code>src</code> 切片中的项复制（可能是重叠）到 <code>dst</code> 切片中，如果空间不够则截断；或者将字符串字节 <code>s</code> 复制到 <code>[]byte</code> 类型的 <code>b</code> 中（参见 4.2.3 节）
<code>delete(m, k)</code>	从映射 <code>m</code> 中删除其键为 <code>k</code> 的项，如果键为空则什么都不做（参见 4.3 节）
<code>imag(cx)</code>	作为 <code>float64</code> 类型的 <code>complex128</code> 类型数据的虚部（参见 2.3.2 节）
<code>len(x)</code>	切片 <code>x</code> 的长度，或者通道 <code>x</code> 的缓冲区中排队的项的数量，或者数组（或者所指向数组）的长度，或者一个映射 <code>x</code> 中项的个数，或者字符串 <code>x</code> 中字节的个数。同时参考 <code>cap()</code> （参见 4.2.3 节）
<code>make(T)</code>	一个切片、映射或者通道类型 <code>T</code> 的引用。如果给定 <code>n</code> ，那它就是该切片的长度和容量，或者提示一个映射需要多少项，或者一个缓冲区的大小。对于切片而言， <code>n</code> 和
<code>make(T, n)</code>	<code>m</code> 可用于声明长度和容量（参见 4.2 节关于切片的内容，参见 4.3 节关于映射的内容，以及第 7 章关于通道的内容）。
<code>make(T, n, m)</code>	
<code>new(T)</code>	一个指向类型 <code>T</code> 的值指针（参见第 4.1 节）
<code>panic(x)</code>	抛出一个运行时异常，其值为 <code>x</code> （参见 5.5.1 节）
<code>real(cx)</code>	类型为 <code>complex128</code> 的 <code>cx</code> 值的实部，是一个 <code>float64</code> 类型值（参见 2.3.2.1 节）
<code>recover()</code>	捕获一个运行时异常（参见 5.5.1 节）

括号放置的美学，通常引来无限多的讨论，但Go语言中不会。这部分是因为自动插入的分号限制了左括号的放置，部分是因为许多Go语言的用户使用gofmt程序将Go代码标准格式化。事实上，Go标准库中的所有源代码都使用了gofmt，这就是为什么这些代码有一个如此紧凑而一致的结构的原因，虽然这些是许多不同程序员的工作 [2]。

Go语言支持表2-4中所列的++（递增）和--（递减）操作符。它们都是后置操作符，也就是说，它们必须跟随在一个它们所作用的操作数后面，并且它们没有返回值。这些限制使得该操作符不能用于表达

式，也意味着不能用于语意不明的上下文中。例如，我们不能将该操作符用于一个函数的语句中，或者在Go语言中写出类似*i=i++*这样的代码（虽然我们能够在C和C++中这样做，其中其结果是未定义的）。

赋值通过使用 = 赋值操作符来完成。变量可以使用 = 和一个 **var** 连接起来创建和赋值。例如，`var x int = 5` 创建了一个 **int** 型的变量 **x**，并将其赋值为 5。（使用 `var x int = 5` 或者 `x := 5` 所达到的目的完全一样。）被赋值变量的类型必须与其所赋的值的类型相兼容。如果使用了 = 而没有使用 **var** 关键字，那么其左边的变量必须是已存在的。可以为多个逗号分隔的变量赋值，我们也可以使用空标识符 (`_`) 来接受赋值，它与任意类型兼容，并且会将赋给它的值忽略。多重赋值使得交换两个变量之间的数据变得更加简单，它不需要显式地指明一个临时变量，例如 `a, b = b, a`。

快速声明操作符 (`:=`) 用于同时在一个语句中声明和赋值一个变量。多个逗号分隔的变量用法大多数情况下跟 = 赋值操作符一样，除了必须至少有一个非空变量为新的。如果有一个变量已经存在了，它就会被直接赋值，而不会新建一个变量，除非该 `:=` 操作符位于作用域的起始处，如 `if` 或者 `for` 语句中的初始化语句（参见 5.2.1 节和 5.3 节）。

```
a, b, c := 2, 3, 5
for a := 7; a < 8; a++{ // a无意间覆盖了外部a的值
    b := 11             // b无意间覆盖了外部b的值
    c = 13              // c为外部的c值  ✓
    fmt.Printf( " inner: a → %d b → %d c → %d\n " , a, b, c)
}
fmt.Printf( " outer: a → %d b → %d c → %d\n " , a, b, c)
```

```
inner: a → 7 b → 11 c → 13
outer: a → 2 b → 3 c → 13
```

这个代码片段展示了:= 操作符是如何创建“影子”变量的。在上面的代码中，for 循环里面，变量a和b覆盖了外部作用域中的变量，虽然合法，但基本上可确定是一个失误。另一方面，上面代码只创建了一个变量c（在外部作用域中），因此它的使用是正确的，并且也是所预期的。我们马上会看到，覆盖其他变量的变量可以很方便，但是粗心地使用可能会引起问题。

正如我们将在后面章节所讨论的，我们可以在有一到多个命名返回值的函数中写无需声明返回值的return语句。这种情况下，返回值将是命名的返回值，它们在函数入口处被初始化为其类型的零值，并且可以在函数体中通过赋值语句来改变它们。

```
func shadow() (err error) { // 该函数不能编译
    x, err := check1() // 创建x，并对err进行赋值
    if err != nil {
        return // 正确地返回err
    }
    if y, err := check2(x); err != nil { // 创建了变量y和一个内部err变量
        return // 内部err变量覆盖了外部err变量，因此错误地返回了nil
    } else {
        fmt.Println(y)
    }
    return // 返回nil
}
```

在shadow()函数的第一个语句中，创建了变量x并将其赋值。但是err变量只是简单地将其赋值，因为它已经被声明为 shadow()函数的返回值了。这之所以能够工作，是因为:=操作符必须至少创建一个非空的变量，而该条件在这里能够满足。因此，如果err变量非空，就会正确地返回。

一个if语句的简单语句（即跟在if后面且在条件之前的可选语句）创建了一个新的作用域（参见5.2.1节）。因此，变量y和err都被重新创建了，后者是一个影子变量。如果err为非空，则返回外部作用域中的err（即声明为shadow()函数返回值的err变量），其值为nil，因为调用check1()函数的时候它被赋值了，而调用check2()的时候，赋值的是err的影子变量。

所幸的是，函数的影子变量问题只是个幻影，因为在我们使用裸的return语句而此时又有任一返回值被影子变量覆盖时，Go编译器会给出一个错误消息并中止编译。因此，该函数无法通过编译。

一个简单的办法是在函数开始处声明变量（例如var x, y int或者x, y := 0, 0），然后把调用 check1()和调用 check2()函数时的:= 换成 =。（关于该方法的一个例子，请看自定义的americanise()函数。）

另一个解决方法是使用一个非命名的返回值。这迫使我们返回一个显式的值，因此在本例中，前两个语句的返回值都是return err（每一个语句返回一个不同的但都是正确的err值），同时最后一个返回语句为return nil。

5.1.1 类型转换

Go语言提供了一种在不同但相互兼容的类型之间相互转换的方式，并且这种转换非常有用并且安全。非数值类型之间的转换不会丢失精度。但是对于数值类型之间的转换，可能会发生丢失精度或者其他问题。例如，如果我们有一个x := uint16(65000)，然后使用转换y := int16(x)，由于x超出了int16的范围，y的值被毫无悬念地设置成-536，这也可能不是我们所想要的。

下面是类型转换的语法：

```
resultOfType := Type(expression)
```

对于数字，本质上讲我们可以将任意的整型或者浮点型数据转换成别的整型或者浮点型（如果目标类型比源类型小，则可能丢失精度）。同样的规则也适用于 `complex128`和`complex64`类型之间的转换。我们已经在2.3节讲解了数字转换的内容。

一个字符串可以转换成一个`[]byte`（其底层为 UTF-8的字节）或者一个`[]rune`（它的Unicode码点），并且`[]byte`和`[]rune`都可以转换成一个字符串类型。单个字符是一个`rune`类型数据（即 `int32`），可以转换成一个单字符的字符串。字符串和字符的类型转换的内容已在第3章中阐述过（参见表3-2、表3-8和表3-9）。

让我们看一个更加直观的小例子，它从一个简单的自定义类型开始。

```
type StringSlice []string
```

该类型也有一个自定义的`StringSlice.String()` 函数（没给出），它返回一个表示一个字符串切片的字符串，该字符串切片以组合字面量语法的形式创建了自定义的`StringSlice`类型。

```
fancy := StringSlice( " Lithium " , " Sodium " , " Potassium " , "
Rubidium " )
fmt.Println(fancy)
plain := []string(fancy)
fmt.Println(plain)
StringSlice{ " Lithium " , " Sodium " , " Potassium " , " Rubidium
" }
[Lithium Sodium Potassium Rubidium]
```

`StringSlice`变量`fancy`使用它自身的`StringSlice.String()`函数打印。但一旦我们将其转换成一个普通的`[]string`切片，那就像任何其他`[]string`一样被打印了。（创建带自身方法的自定义类型的内容将在第6章提到。）

如果表达式与类型Type的底层类型一样，或者如果表达式是一个可以用类型Type表达的无类型常量，或者如果Type是一个接口类型并且该表达式实现了Type接口，那么将一种类型的数据转换成其他类型也是可以的 [3]。

5.1.2 类型断言

一种类型的方法集是一个可以被该类型的值调用的所有方法的集合。如果该类型没有方法，则该集合为空。Go语言的interface{}类型用于表示空接口，即一个方法集为空集的类型。由于每一种类型都有一个方法集包含空的集合（无论它包含多少方法），一个interface{}的值可以用于表示任意Go类型的值。此外，我们可以使用类型开关、类型断言或者Go语言的reflect包的类型检查（参见9.4.9节）将一个interface{}类型的值转换成实际数据的值（参见5.2.2.2节） [4]。

在处理从外部源接收到的数据、想创建一个通用函数及在进行面向对象编程时，我们会需要使用interface{}类型（或自定义接口类型）。为了访问底层值，有一种方法是使用下面中提到的一种语法来进行类型断言：

`resultOfType, boolean := expression.(Type) // 安全类型断言`

`resultOfType := expression.(Type) // 非安全类型断言，失败时panic()`

成功的安全类型断言将返回目标类型的值和标识成功的true。如果安全类型断言失败（即表达式的类型与声明的类型不兼容），将返回目标类型的零值和false。非安全类型断言要么返回一个目标类型的值，要么调用内置的panic()函数抛出一个异常。如果异常没有被恢复，那么该函数会导致程序终止。（异常的抛出和恢复的内容将在后面阐述，参见5.5节。）

这里有个小程序用来解释用到的语法。

```

var i interface{} = 99
var s interface{} = []string{ " left " , " right " }
j := i.(int) // j是int类型的数据（或者发生了一个panic()）
fmt.Printf( " %T → %d\n " , j, j)
if i, ok := i.(int); ok {
    fmt.Printf( " %T → %d\n " , i, j) // i是一个int类型的影子变量
}
if s, ok := s.([]string); ok {
    fmt.Printf( " %T → %q\n " , s, s) // s是一个[]string类型的影子变量
}
int → 99
int → 99
[]string → [ " left " " right " ]

```

做类型断言的时候将结果赋值给与原始变量同名的变量是很常见的事情，即使用影子变量。同时，只有在我们希望表达式是某种特定类型的值时才使用类型断言。（如果目标类型可以是许多类型之一，我们可以使用类型开关，参见5.2.2.2节。）

注意，如果我们输出原始的*i*和*s*变量（两者都是`interface{}`类型），它们可以以`int`和`[]string`类型的形式输出。这是因为当`fmt`包的打印函数遇到`interface{}`类型时，它们会足够智能地打印实际类型的值。

5.2 分支

Go语言提供了3种分支语，即`if`、`switch`和`select`，后者将在后面深入讨论（参见5.4节）。分支效果也可以通过使用一个映射来达到，它

的键可以用于选择分支，而它的值是对应的要调用的函数，我们会在本章末尾看到更多细节（参见5.6.5节）。

5.2.1 if语句

Go语言的if语句语法如下：

```
if optionalStatement1; booleanExpression1 {  
    block1  
} else if optionalStatement2; booleanExpression2 {  
    block2  
} else {  
    block3  
}
```

一个if语句中可能包含零到多个**else if**子句，以及零到多个**else**子句。每一个代码块都由零到多个语句组成。

语句中的大括号是强制性的，但条件判断中的分号只有在可选的声明语句**optionalStatement1**出现的情况下才需要。该可选的声明语句用Go语言的术语来说叫做“简单语句”。这意味着它只能是一个表达式、发送到通道（使用**<-**操作符）、增减值语句、赋值语句或者短变量声明语句。如果变量是在一个可选的声明语句中创建的（即使用**:=**操作符创建的），它们的作用域会从声明处扩展到if语句的完成处，因此它们在声明它们的if或者**else if**语句以及相应的分支中一直存在着，直到该if语句的末尾。

布尔表达式必须是**bool**型的。Go语言不会自动转换非布尔值，因此我们必须使用比较操作符。例如，**if i == 0**。（布尔类型和比较操作符参见表2-3。）

我们已经看过了使用if语句的大量例子，在本书的后续章节中将看到更多。不过，让我们再看两个小例子，第一个演示了可选简单语句的用处，第二个解释了Go语言中if语句的习惯用法。

// 经典用法	// 啰嗦用法
if $\alpha := \text{compute}()$; $\alpha < 0$ {	{
fmt.Printf(" (%d)\n " , - α)	$\alpha := \text{compute}()$
} else {	if $\alpha < 0$ {
fmt.Println(α)	fmt.Printf("
(%d)\n " , - α)	
}	} else {
	fmt.Println(α)
	}
	}

这两段代码的输出一模一样。右边的代码必须使用额外的大括号来限制变量 α 的作用域，然而左边的代码中的if语句自动地限制了变量的作用域。

第二个关于if语句的例子是ArchiveFileList()函数，它来自于archive_file_list示例（在文件archive_file_list/archive_file_list.go中）。随后，我们会使用该函数的实现来对比if和switch语句。

```
func ArchiveFileList(file string) ([]string, error) {
    if suffix := Suffix(file); suffix == ".gz" {
        return GzipFileList(file)
    } else if suffix == ".tar" || suffix == ".tar.gz" || suffix == ".tgz"
    {
        return TarFileList(file)
    } else if suffix == ".zip" {
        return ZipFileList(file)
    }
}
```

```

    }
    return nil, errors.New( " unrecognized archive " )
}

```

该函数读取一个从命令行指定的文件，对于那些可以处理的压缩文件（.gz、.tar、.tar.gz、.zip），它会打印压缩文件的文件名，并以缩进格式打印该压缩文件所包含文件的列表。

第一个if语句中声明的suffix变量的作用域扩展到了整个if...else if...语句中，因此它在每一个分支中都是可见的，就像前例中的 α 变量一样。

该函数本可以在末尾添加一个else语句，但在Go语言中使用这里所给的结构是非常常用的：一个if语句带零到多个else if语句，其中每一个分支都带有一个return语句，随后紧接的是一个return语句而非一个包含return语句的else分支。

```

func Suffix(file string) string {
    file = strings.ToLower(filepath.Base(file))
    if i := strings.LastIndex(file, "."); i > -1 {
        if file[i:] == ".bz2" || file[i:] == ".gz" || file[i:] == ".xz" {
            if j := strings.LastIndex(file[:i], ".");
                j > -1 && strings.HasPrefix(file[j:], ".tar") {
                return file[j:]
            }
        }
        return file[i:]
    }
    return file
}

```

为了完整性考虑，这里也提供了**Suffix()**函数的实现。它接受一个文件名（可能包含路径），返回其小写的后缀名（也叫扩展名），即文件名中从点号开始的最后部分。如果一个文件名没有点号，则将文件名返回（路径除外）。如果文件名以**.tar.bz2**、**.tar.gz** 或者**.tar.xz**结尾，则这些就是返回的后缀。

5.2.2 switch语句

Go语言中有两种类型的**switch**语句：表达式开关（**expression switch**）和类型开关（**type switch**）。表达式开关语句对于C、C++和Java程序员来说比较熟悉，然而类型开关语句是Go语言专有的。两者在语法上非常相似，但不同于C、C++和Java的是，Go语言的**switch**语句不会自动地向下贯穿（因此不必在每一个**case**子句的末尾都添加一个**break**语句）。相反，我们可以在需要的时候通过显式地调用**fallthrough**语句来这样做。

5.2.2.1 表达式开关

Go语言的表达式开关（**switch**）语法如下：

```
switch optionalStatement; optionalExpression {  
case expressionList1: block1  
...  
case expressionListN: blockN  
default: blockD  
}
```

如果有可选的声明语句，那么其中的分号是必要的，无论后面可选的表达式语句是否出现。每一个块由零到多个语句组成。

如果**switch**语句未包含可选的表达式语句，那么编译器会假设其表达式值为**true**。可选的声明语句与**if**语句中使用的简单语句是相同类型

的。如果变量都是在可选的声明语句中创建的（例如，使用`:=`操作符），它们的作用域将会从其声明处扩展到整个`switch`语句的末尾处。因此它们在每个`case`语句和`default`语句中都存在，并在`switch`语句的末尾处消失。

将`case`语句排序最有效的办法是，从头至尾按最有可能到最没可能的顺序列出来，虽然这只有在有很多`case`子句并且该`switch`语句重复执行的情况下才显得重要。由于`case`子句不会自动地向下贯穿，因此没必要在每一个`case`语句块的末尾都加上一个`break`语句。如果需要`case`语句向下贯穿，我们只需简单地使用一个`fallthrough`语句。`default`语句是可选的，并且如果出现了，可以放在任意地方。如果没有一个`case`表达式匹配，则执行给出的`default`语句；否则程序将从`switch`语句之后的语句继续往下执行。

每一个`case`语句必须有一个表达式列表，其中包含一个或者多个分号分隔的表达式，其类型与`switch`语句中的可选表达式类型相匹配。如果没有给出可选的表达式，编译器会自动将其设置为`true`，即一个布尔类型，这样每一个`case`子句中的表达式的值就必须是一个布尔类型。

如果一个`case`或者`default`语句有一个`break`语句，`switch`语句的执行会被立即跳出，其控制权被交给`switch`语句后面的语句，或者如果`break`语句声明了一个标签，控制权就会交给声明标签处的最里层`for`、`switch`或者`select`语句。

这里有个关于`switch`语句的非常简单的例子，它没有可选的声明和可选表达式。

```
func BoundedInt(minimum, value, maximum int) int {  
    switch {  
    case value < minimum:  
        return minimum  
    case value > maximum:
```

```

    return maximum
}
return value
}

```

由于没有可选的表达式语句，编译器会将表达式语句的值设为 `true`。这意味着 `case` 语句中的每一个表达式都必须计算为布尔类型。这里两个表达式语句都使用了布尔比较操作符。

```

switch {
case value < minimum:
    return minimum
case value > maximum
    return maximum
default:
    return value
}
panic( " unreachable " )

```

这是上面 `BoundedInt()` 函数的一种替代实现。其 `switch` 语句现在包含了每一种可能的情况，因此控制权永远不会到达 `switch` 语句的末尾。然而，Go 语言希望在函数的末尾出现一个 `return` 语句或者 `panic()`，因此我们使用了后者来更好地表达函数的语意。

前面节中的 `ArchiveFileList()` 函数使用了一个 `if` 语句来决定调用哪个函数。这里有一个原始的基于 `switch` 语句的版本。

```

switch suffix := Suffix(file); suffix { || 原始的非经典用法
case ".gz" :
    return GzipFileList(file)
case ".tar" :
    fallthrough

```



```

case ".tar.gz" :
    fallthrough
case ".tgz" :
    return TarFileList(file)
case ".zip" :
    return ZipFileList(file)
}

```

`switch`语句同时有一个声明语句和一个表达式语句。本例中表达式语句是`string`类型，因此每一个 `case` 语句的表达式列表必须包含一个或者多个以逗号分隔的字符串才能匹配。我们使用了`fallthrough`语句来保证所有的`tar`类型文件都使用同一个函数来执行。

变量`suffix`的作用域从声明处扩展至每一个`case`子句（如果有`default`，其作用域也会扩展至`default`子句）中，同时在`switch`语句的末尾处结束，因为从那之后`suffix`变量就不再存在了。

```

switch Suffix(file) { // 经典用法
case ".gz" :
    return GzipFileList(file)
case ".tar", ".tar.gz", ".tgz" :
    return TarFileList(file)
case ".zip" :
    return ZipFileList(file)
}

```

这里有个更加紧凑也更加实用的使用`switch`的版本。与使用一个声明和一个表达式语句不同的是，我们只是简单地使用一个表达式：一个返回字符串的`Suffix()`的函数。同时，我们也不使用 `fallthrough` 语句来处理所有的`tar` 文件，而是使用逗号分隔的所有能够匹配的文件后缀来作为`case`语句的表达式列表。

Go语言的表达式switch语句比C、C++以及Java中的类似语句都更有用，很多情况下可以用于代替if语句，并且还更紧凑。

5.2.2.2 类型开关

注意，我们之前提到过类型断言（参见5.1.2节），当我们使用interface{}类型的变量时，我们常常需要访问其底层值。如果我们知道其类型，就可以使用类型断言，但如果其类型可能是许多可能类型中的一种，那我们就可以使用类型开关语句。

Go语言的类型开关语法如下：

```
switch optionalStatement; typeSwitchGuard {  
  case typeList1: block1  
  ...  
  case typeListN: blockN  
  default: blockD  
}
```

可选的声明语句与表达式开关语句和if语句中的一样。同时这里的case语句与表达式切换语句中的case语句工作方式也一样，不同的是这里列出一个或者以多个逗号分隔的类型。可选的default子句和fallthrough语句与表达式切换语句中的工作方式也一样，通常，每一个块也包含零到多条语句。

类型开关守护（guard）是一个结果为类型的表达式。如果表达式是使用:=操作符赋值的，那么创建的变量的值为类型开关守护表达式中的值，但其类型则决定于case子句。在一个列表只有一个类型的case子句中，该变量的类型即为该类型；在一个列表包含两个或者更多个类型的case子句中，其变量的类型则为类型开关守护表达式的类型。

这种类型开关语句所支持的类型测试对于面向对象程序员来说可能比较困惑，因为他们更依赖于多态。Go语言在一定程度上可以通过

鸭子类型支持多态（将在第6章看到），但尽管如此，有时使用显式的类型测试是更为明智的选择。

这里有个例子，显示了我们如何调用一个简单的类型分类函数以及它的输出。

```
classifier(5, -17.9, " ZIP ", nil, true, complex(1, 1))
```

```
param #0 is an int
param #1 is a float64
param #2 is a string
param #3 is nil
param #4 is a bool
param #5's type is unknown
```

`classifier()`函数使用了一个简单的类型开关。它是一个可变参函数，也就是说，它可以接受不定数量的参数。并且由于其参数类型为`interface{}`，所传的参数可以是任意类型的。（本章稍后将讲解可变参函数以及带省略符函数，参见5.6节。）

```
func classifier(items...interface{}) {
    for i, x := range items {
        switch x.(type) {
        case bool:
            fmt.Printf( " param #%d is a bool\n ", i)
        case float64:
            fmt.Printf( " param #%d is a float64\n ", i)
        case int, int8, int16, int32, int64:
            fmt.Printf( " param #%d is an int\n ", i)
        case uint, uint8, uint16, uint32, uint64:
            fmt.Printf( " param #%d is an unsigned int\n ", i)
        case nil:
```

```

        fmt.Printf( " param #%d is nil\n " , i)
    case string:
        fmt.Printf( " param #%d is a string\n " , i)
    default:
        fmt.Printf( " param #%d's type is unknow\n " , i)
    }
}
}

```

这里使用的类型开关守护与类型断言里的格式一样，即 `variable.(Type)`，但是使用 `type` 关键字而非一个实际类型，以用于表示任意类型。

有时我们可能想在访问一个 `interface{}` 的底层值的同时也访问它的类型。我们马上会看到，这可以通过将类型开关守护进行赋值（使用 `:=` 操作符）来达到这个目的。

类型测试的一个常用案例是处理外部数据。例如，如果我们解析JSON格式的数据，我们必须将数据转换成相对应的Go语言数据类型。这可以通过使用Go语言的 `json.Unmarshal()` 函数来实现。如果我们向该函数传入一个指向结构体的指针，该结构体又与该JSON数据相匹配，那么该函数就会将JSON数据中对应的数据项填充到结构体的每一个字段。但是如果我们先并不知道JSON数据的结构，那么就不能给 `json.Unmarshal()` 函数传入一个结构体。这种情况下，我们可以给该函数传入一个指向 `interface{}` 的指针，这样 `json.Unmarshal()` 函数就会将其设置成引用一个 `map[string]interface{}` 类型值，其键为JSON字段的名称，而值为对应的保存为 `interface{}` 的值。

这里有个例子，给出了如何反序列化一个其内部结构未知的原始JSON对象，如何创建和打印JSON对象的字符串表示。

```

MA := []byte('{ " name " : " Massachusetts " , " area " : 27336, "
water " : 25.7, " senators " :
    [ " John Kerry " , " Scott Brown " ]}')
var object interface{}
if err := json.Unmarshal(MA, &object); err != nil {
    fmt.Println(err)
} else {
    jsonObject := object.(map[string]interface{}) ①
    fmt.Println(jsonObjectAsString(jsonObject))
}
{ " senators " : [ " John Kerry " , " Scott Brown " ], " name " : "
Massachusetts " ,
    " water " : 25.700000, " area " : 27336.000000}

```

如果反序列化时未发生错误，则 `interface{}` 类型的 `object` 变量就会指向一个 `map[string]interface{}` 类型的变量，其键为 JSON 对象中字段的名称。`jsonObjectAsString()` 函数接收一个该类型的映射，同时返回一个对应的 JSON 字符串。我们使用一个未检查的类型断言语句（标识①）来将一个 `interface{}` 类型的对象转换成 `map[string] interface{}` 类型的 `jsonObject` 变量。（注意，为了适应书页的宽度，这里给出的输出切分成了两行。）

```

func jsonObjectAsString(jsonObject map[string]interface{}) string{
    var buffer bytes.Buffer
    buffer.WriteString( " { " )
    comma := " , "
    for key, value := range jsonObject{
        buffer.WriteString(comma)
        switch value := value.(type){ // 影子变量 ①

```

```

case nil: ②
    fmt.Fprintf(&buffer, " %q: null ", key)
case bool:
    fmt.Fprintf(&buffer, " %q: %t ", key, value)
case float64:
    fmt.Fprintf(&buffer, " %q: %f ", key, value)
case string:
    fmt.Fprintf(&buffer, " %q: %q ", key, value)
case []interface{}:
    fmt.Fprintf(&buffer, " %q: [ ", key)
    innerComma := " "
    for _, s := range value{
        if s, ok := s.(string); ok { ||影子变量③
            fmt.Fprintf(&buffer, " %s%q ", innerComma, s)
            innerComma = ", "
        }
    }
    buffer.WriteString( " ] " )
}
comma = " , "
}
buffer.WriteString( " } " )
return buffer.String()
}

```

该函数将一个JSON对象转换成用一个映射来表示，同时返回一个对应的JSON格式中对象的数据的字符串表示。表示JSON对象的映射里

的JSON数组使用`[]interface{}`类型来表示。关于JSON数组，该函数做了一个简化的假设：它假设数组中只包含字符串类型的项。

为了访问数据，我们在`for...range`（参见5.3节）循环来访问映射的键和值，同时使用类型开关来获得和处理每种不同类型的值。类型开关守护（①）将其值（`interface{}`类型）赋值给一个新的`value`变量，其类型与其相匹配的`case`子句的类型相同。在这种情况下使用影子变量是个明智的选择（虽然我们可以轻松地创建一个新的变量）。因此，如果`interface{}`值的类型是布尔型，其内部值为布尔值，那么将匹配第二个`case`子句，其他`case`子句的情况也类似。

为了将数据写回缓冲区，我们使用了`fmt.Fprintf()`函数，因为这个函数比`buffer.WriteString(fmt.Sprintf(...))`（②）函数来得方便。`fmt.Fprintf()`函数将数据写入到其第一个`io.Writer`类型的参数。虽然`bytes.Buffer`不是`io.Writer`，但`*bytes.Buffer`却是一个`io.Writer`，因此我们传入`buffer`的地址。这些内容将在第6章详细阐述。简而言之，`io.Writer`是一个接口，任何提供了`Write()`方法的值都可以满足该接口。`bytes.Buffer.Write()`方法需要一个指针类型的接收器（即一个`*bytes.Buffer`而非一个`bytes.Buffer`值），因此只有`*bytes.Buffer`才能够满足该接口，这也意味着我们必须将`buffer`的地址传入`fmt.Fprintf()`函数，而非`buffer`本身。

如果该JSON对象包含JSON数组，我们使用`for...range`循环来迭代`[]interface{}`数组的每一个项，同时也使用已检查的类型断言来判断（③），这样就能保证只有在数据为字符串类型时我们才将其添加到输出结果中。我们再一次使用了影子变量（这次是字符串类型的`s`），因为我们需要的不是接口，而是该接口所引用的值。（类型断言的内容我们已经讲过，参见5.1.2节。）当然，如果我们事先知道原始JSON对象的结构，我们可以很大程度上简化代码。我们可以使用一个结构

体来保存数据，然后使用一个方法以字符串的形式将其输出。下面是在这种情况下反序列化并将其数据输出的例子。

```
var state State
if err := json.Unmarshal(MA, &state); err != nil {
    fmt.Println(err)
}
fmt.Println(state)
{ " name " : " Massachusetts " , " area " : 27336, " water " :
25.700000,
  " senators " : [ " John Kerry " , " Scott Brown " ] }
```

这段代码看起来跟之前的代码很像。然而，这里不需要 `jsonObjectAsString()` 函数，相反我们需要定义一个 `State` 类型和一个对应的 `State.String()` 方法。（同样地，我们将其输出结果分行以适应书页的宽度。）

```
type State struct{
    Name      string
    Senators []string
    Water      float64
    Area       int
}
```

该结构体与我们之前所看到的近似。然而请注意，这里每个字段的起始字符必须以大写字母开头，这样就能够将其导出（公开），因为 `json.Unmarshal()` 函数只能填充可导出的字段。同时，虽然Go语言的 `encoding/json` 包并不区分不同的数据类型（它会把所有JSON的数字当成 `float64` 类型），但 `json.Unmarshal()` 函数足够聪明，会自动填充其他数据类型的字段。

```
func (state State) String() string{
```



```

var senators []string
for _, senator := range state.Senators{
    senators := append(senators, fmt.Sprintf( " %q " , senator))
}
return fmt.Sprintf(
    '{ " name " : %q, " area " : %d, " water " : %f, " senators " :
[%s]}',
    state.Name, state.Area, state.Water, strings.Join(senators, " , " ))
}

```

该方法返回一个表示State值的JSON字符串。

大部分 Go 程序应该都不需要类型断言和类型开关，即使需要，应该也很少用到。其中一个使用案例是，我们传入一个满足某个接口的值，同时想检查下它是否满足另外一个接口。（该主题将在第6章阐述，例如6.5.2节。）另一个使用案例是，数据来自于外部源但必须转换成Go语言的数据类型。为了简化维护，最好总是将这些代码与其他程序分开。这样就使得程序完全地工作于 Go语言的数据类型之上，也意味着任何外部源数据的格式或类型改变所导致的代码维护工作可以控制在小范围内。

5.3 for循环语句

Go语言使用两种类型的for 语句来进行循环，一种是无格式的for 语句，另一种是for...range语句。下面是它们的语法：

```

for { //无限循环
    block
}

```

```

for booleanExpression { // while循环
    block
}
for optionalPreStatement; booleanExpress; optionalPostStatement{ // ①
    block
}
for index, char := range aString{ //一个字符一个字符地迭代一个字符
串 ②
    block
}
for index := range aString{ // 一个字符一个字符地迭代一个字符串
③
    block // char, size := utf8.DecodeRuneInString(aString[index:])
}
for index, item := range anArrayOfSlice { // 数组或者切片迭代 ④)
    block
}
for index := range anArrayOfSlice { // 数组或者切片迭代 ⑤
    block // item := anArrayOfSlice[index]
}
for key, value := range aMap{ // 映射迭代 // ⑥
    block
}
for key := range aMap { // 映射迭代 // ⑦
    block // value := aMap[key]
}
for item := range aChannel { // 通道迭代

```

```
    block
}
```

for循环中的大括号是必须的，但分号只在可选的前置或者后置声明语句都存在的时候才需要（①），两个声明语句都必须是简短的声明语句。如果变量是在一个可选的声明语句中创建的，或者用来保存一个 **range** 子句中产生的值（例如，使用**:=** 操作符），那么它们的作用域就会从其声明处扩展到**for**语句的末尾。

在无格式的**for**循环语法（①）中，布尔表达式的值必须是**bool**类型的，因为Go语言不会自动转换非**bool**型的值。（布尔表达式和比较操作符的内容已在之前的表2-3中列出。）第二个**for...range**循环迭代一个字符串的语法（③）给出了字节偏移的索引。对于一个7位的ASCII字符串s，如其值为“XabYcZ”，该语句产生的输出为0、1、2、3、4和5。但是对于一个UTF-8的字符串类型s，例如其值为“XαβYγZ”，则产生的索引值为0、1、3、5、6、8。第一个迭代字符串的**for...range**循环语法（②）在大多数情况下都比第二种语法（③）方便。

对于非空切片或者索引而言，第二个迭代数组或者切片的**for...range**循环语法（⑤）获取索引从0到len(slice) - 1的项。该语法与第一个迭代数组或者切片的语法（④）都非常有用。这两个语法能够解释为什么Go程序中更少使用普通的**for**循环（①）。

迭代映射的键-值对（⑥）或键（⑦）的**for...range**循环以任意顺序的形式得到映射中的项或者键。如果需要有序的映射，解决方案之一是使用第二种语法（⑦）创建一个由键组成的切片，然后将切片排序。我们已经在前面章节中看过一个相关的例子（参见4.3.4节）。另一种解决方案是优先使用一个有序数据结构。例如，一个有序映射。我们将在下章看一个类似的例子（参见6.5.3节）。

如果以上语法（②～⑦）作用于一个空字符串、数组、切片或映射，那么**for**循环就什么也不做，控制流程将从下一条语句继续。

一个for循环可以随时使用一个break语句来终止，这样控制权将传递给for循环语句的下一条语句。如果 break 语句声明了一个标签，那么控制权就会进入包含该标签的最内层for、switch或者select语句中。也可以通过使用一个continue语句来使得程序的控制权回到for循环的条件或者范围子句，以进行下一次迭代（或者结束循环）。

我们已经在看到过很多for语句的使用案例，其中包含for...range循环、无限循环以及在Go语言中使用得不是很多的普通for循环（因为其他循环更为方便）。当然，在本书的后续章节以及本章的后面节中，我们也会看到很多使用for循环的例子，因此这里我们就只看一个小例子。

假设我们有一个二维切片（即其类型为[][]int），想要从中搜索看看是否包含某个特定的值。这里有两种搜索的方法。两者都使用第二种遍历数组或切片的for...range循环语法（⑤）。

<pre>found := false for row := range table { for column := range table[row] { if table[row][column] == x { found = true break } } if found { break } }</pre>	<pre>found := false FOUND: for row := range table { for column := range table[row] { if table[row][column] == x { found = true break FOUND } } }</pre>
--	--

标签是一个后面带一个冒号的标识符。这两个代码段的功能一样，但是右边的代码比左边的代码更加简短和清晰，因为一旦成功搜索到目标值(x)，它就会使用一个声明了一个标签的break子句跳转到外层循环。如果我们的循环嵌套得很深（例如，迭代一个三维的数据），使用带标签的中断语句的优势就更加明显。

标签可以作用于for、switch以及select语句。break和continue语句都可以声明标签，并且都可用于for循环里面。同时，也可以在switch和

`select`语句里面使用`break`语句，无论是裸的`break`语句还是声明了一个标签的`break`语句。

标签也可以独立出现在程序中，它们可能用做`goto`语句的目标（使用`goto label`语法）。如果一个 `goto` 语句跳过了任何创建变量的语句，则程序的行为是未定义的。幸运的话程序会崩溃，但它也可能继续运行并输出错误的结果。一个使用 `goto` 语句的案例是用于自动生成代码，因为在这种情况下`goto`语句非常方便，并且无需顾虑意大利面式代码问题（`spaghetti code`，指代码的控制结构特别复杂难懂）。虽然在写本书时有超过30个Go语言的源代码文件中使用了`goto`语句，但本书的例子中不会出现`goto`语句，我们提倡避免它 [5]。

5.4 通信和并发语句

Go语言的通信与并发特性将在第7章讲解，但是为了过程式编程讲解的完整性，我们在这里描述下它的基本语法。

`goroutine` 是程序中与其他`goroutine` 完全相互独立而并发执行的函数或者方法调用。每一个Go 程序都至少有一个`goroutine`，即会执行`main`包中的`main()`函数的主`goroutine`。`goroutine`非常像轻量级的线程或者协程，它们可以被大批量地创建（相比之下，即使是少量的线程也会消耗大量的机器资源）。所有的`goroutine`共享相同的地址空间，同时Go语言提供了锁原语来保证数据能够安全地跨`goroutine`共享。然而，Go语言推荐的并发编程方式是通信，而非共享数据。

Go语言的通道是一个双向或者单向的通信管道，它们可用于在两个或者多个`goroutine`之间通信（即发送和接收）数据。

在`goroutine`和通道之间，它们提供了一种轻量级（即可扩展的）并发方式，该方式不需要共享内存，因此也不需要锁。但是，与所有其

他的并发方式一样，创建并发程序时务必要小心，同时与非并发程序相比，对并发程序的维护也更有挑战。大多数操作系统都能够很好地同时运行多个程序，因此利用好这点可以降低维护的难度。例如，将多份程序（或者相同程序的多份副本）的每一个操作作用于不同的数据上。优秀的程序员只有在其带来的优点明显超过其所带来的负担时才会编写并发程序。

goroutine使用以下的go语句创建：

```
go function(arguments)
```

```
go func(parameters) { block } (arguments)
```

我们必须要么调用一个已有的函数，要么调用一个临时创建的匿名函数。与其他函数一样，该函数可能包含零到多个参数，并且如果它包含参数，那么必须像其他函数调用一样传入对应的参数。

被调用函数的执行会立即进行，但它是在另一个goroutine上执行，并且当前goroutine的执行（即包含该go语句的goroutine）会从下一条语句中立即恢复。因此，执行一个go语句之后，当前程序中至少有两个goroutine在运行，其中包括原始的goroutine（初始的主goroutine）和新创建的goroutine。

少数情况下需要开启一串的goroutine，并等待它们完成，同时也不需要通信。然而，在大多数情况下，goroutine之间需要相互协作，这最好通过让它们相互通信来完成。下面是用于发送和接收数据的语法：

```
channel <- value           // 阻塞发送
```

```
<-channel                 // 接收并将其丢弃
```

```
x := <-channel            // 接收并将其保存
```

```
x, ok := <-channel       // 功能同上，同时检查通道是否已关闭
```

或者是否为空

非阻塞的发送可以使用select语句来达到，或者在一些情况下使用带缓冲的通道。通道可以使用内置的make()函数通过以下语法来创建：

`make(chan Type)`

`make(chan Type, capacity)`

如果没有声明缓冲区容量，那么该通道就是同步的，因此会阻塞直到发送者准备好发送和接收者准备好接收。如果给定了一个缓冲区容量，通道就是异步的。只要缓冲区有未使用空间用于发送数据，或还包含可以接收的数据，那么其通信就会无阻塞地进行。

通道默认是双向的，但如果需要我们可以使得它们是单向的。例如，为了以编译器强制的方式更好地表达我们的语义。在第7章中我们将看到如何创建单向的通道，然后在任何适当的时候都使用单向通道。

让我们结合一个小例子理解上文中讨论的语法 [6]。我们将创建返回一个通道的`createCounter()`函数。当我们从中接收数据时，该通道会发送一个 `int` 类型数据。通道返回的第一个值是我们传送给 `createCounter()`函数的值，往后返回的每一个值都比前面一个大1。下面展示了我们如何创建两个独立的 `counter` 通道（每个都在它们自己的 `goroutine` 里执行）以及它们产生的结果。

```
counterA := createCounter(2)      // counterA是chan int类型的
counterB := createCounter(102)    // counterB是chan int类型的
for i := 0; i < 5; i++ {
    a := <-counterA
    fmt.Printf( " (A → %d, B → %d) " , a, <-counterB)
}
fmt.Println()
```

```
(A → 2, B → 102) (A → 3, B → 103) (A → 4, B → 104) (A → 5,
B → 105) (A → 6, B → 106)
```

我们用两种方式展示了如何从通道获取数据。第一种接收方式将获取的数据保存到一个变量里，第二种接收方式将接收的值直接以参数的形式传递给一个函数。

这两个 `createCounter()` 函数的调用是在主 `goroutine` 中进行的，而另外两个由 `createCounter()` 函数创建的 `goroutine` 初始时都被阻塞。在主 `goroutine` 中，只要我们一从这两个通道中接收数据，就会发生一次数据发送，然后我们就能接收其值。然后，发送数据的 `goroutine` 再次阻塞，等待一个新的接收请求。这两个通道是无限的，即它们可以无限地发送数据。（当然，如果我们达到了 `int` 型数据的极限，下一个值就会从头开始。）一旦我们想要接收的五个值都从通道中接收完成，通道将继续阻塞以备后续使用。

如果不再需要了，我们如何清理用于计数器通道的 `goroutine` 呢？这需要让它跳出无限循环，以终止发送数据，然后关闭它们使用的通道。我们将在下一节提供一种方法。当然，第 7 章中我们将深入讨论更多关于并发的内容。

```
func createCounter(start int) chan int{
    next := make(chan int)
    go func(i int) {
        for {
            next <- i
            i++
        }
    }(start)
    return next
}
```

该函数接收一个初始值，然后创建一个通道用于发送和接收 `int` 型数据。然后，它将该初始值传入在一个新的 `goroutine` 中执行的匿名函

数。该匿名函数有一个无限循环，它简单地发送一个int型数据，并在每次迭代中将该int型数据加1。由于通道创建时其容量为0，因此该发送会阻塞直到收到一个从通道中接收数据的请求。该阻塞只会影响匿名函数所在的goroutine，因此程序中剩下的其他goroutine对此一无所知，并且将继续运行。一旦该goroutine被设置为运行状态（当然，从这点来看它会立即阻塞），紧接着该函数的下一条语句会立即执行，将通道返回给其调用者。

有些情况下我们可能有多个goroutine并发执行，每一个goroutine都有其自身通道。我们可以使用select语句来监控它们的通信。

select语句

Go语言的select语句语法如下 [7]：

```
select {  
    case sendOrReceive1: block1  
    ...  
    case sendOrReceiveN: blockN  
    default: blockD  
}
```

在一个 select 语句中，Go语言会按顺序从头至尾评估每一个发送和接收语句。如果其中的任意一语句可以继续执行（即没有被阻塞），那么就从那些可以执行的语句中任意选择一条来使用。如果没有任意一条语句可以执行（即所有的通道都被阻塞），那么有两种可能的情况。如果给出了default语句，那么就会执行default语句，同时程序的执行会从select语句后的语句中恢复。但是如果没有default语句，那么select语句将被阻塞，直到至少有一个通信可以继续下去。

一个select语句的逻辑结果如下所示。一个没有default语句的select语句会阻塞，只有当至少有一个通信（接收或者发送）到达时才完成阻塞。一个包含 default 语句的select语句是非阻塞的，并且会立即执

行，这种情况下可能是因为有通信发生，或者如果没有通信发生就会执行default语句。

为了了解和掌握该语法，让我们来看两个简短的例子。第一个例子有些刻意为之，但能够让我们很好地理解select语句是如何工作的。第二个例子给出了更为符合实际的用法。

```
channels := make([]chan bool, 6)
for i := range channels {
    channels[i] = make(chan bool)
}
go func() {
    for {
        channels[rand.Intn(6)] <- true
    }
}()
```

在上面的代码片段中，我们创建了6个用于发送和接收布尔数据的通道。然后我们创建了一个goroutine，其中有一个无限循环语句，在循环中每次迭代都随机选择一个通道并发送一个true值。当然，该goroutine会立即阻塞，因为这些通道不带缓冲且我们还没从这些通道中接收数据。

```
for i := 0; i < 36; i++ {
    var x int
    select {
    case <-channels[0]:
        x = 1
    case <-channels[1]:
        x = 2
    case <-channels[2]:
```

```

        x = 3
    case <-channels[3]:
        x = 4
    case <-channels[4]:
        x = 5
    case <-channels[5]:
        x = 6
    }
    fmt.Printf( " %d ", x)
}
fmt.Println()

```

```
6 4 6 5 4 1 2 1 2 1 5 5 4 6 2 3 6 5 1 5 4 4 3 2 3 3 3 5 3 6 5 2 2 3 6 2
```

上面代码片段中，我们使用 6 个通道来模拟一个公平骰子的滚动（严格地讲，是一个伪随机的骰子）。其中的`select`语句等待通道发送数据，由于我们没有提供一个`default`语句，该`select`语句会阻塞。一旦有一个或者更多个通道准备好了发送数据，那么程序会以伪随机的形式选择一个`case`语句来执行。由于该`select`语句在一个普通`for`循环内部，它会执行固定数量的次数。

接下来让我们看一个更加实际的例子。假设我们要对两个独立的数据集进行同样的昂贵计算，并产生一系列结果。下面是执行该计算的函数框架。

```

func expensiveComputation(data Data, answer chan int, done chan
bool) {
    // 设置.....
    finished := false
    for !finished {

```

```

        // 计算.....
        answer <- result
    }
    done <- true
}

```

该函数接收需要计算的数据和两个通道。 **answer** 通道用于将每个结果发送回监控代码中，而 **done** 通道则用于通知监控代码计算已经完成。

```

// 设置 .....
const allDone = 2
doneCount := 0
answer $\alpha$  := make(chan int)
answer $\beta$  := make(chan int)
defer func() {
    close(answer $\alpha$ )
    close(answer $\beta$ )
}()
done := make(chan bool)
defer func() { close(done) }()
go expensiveComputation(data1, answer $\alpha$ , done)
go expensiveComputation(data2, answer $\beta$ , done)
for doneCount != allDone {
    var which, result int
    select {
    case result = <-answer $\alpha$ :
        which = ' $\alpha$ '
    case result = <-answer $\beta$ :

```

```

        which = 'β'
    case <-done:
        doneCount++
    }
    if which != 0 {
        fmt.Printf( " %c → %d ", which, result)
    }
}
fmt.Println()

```

```

α → 3 β → 3 α → 0 β → 9 α → 0 β → 2 α → 9 β → 3 α → 6 β → 1 α → 0
β → 8 α → 8 β → 5 α → 0 β → 0 α → 3

```

上面这些代码设置了通道，并开始执行计算，监控进度，然后在程序的末尾进行清理。以上代码没出现一个锁。

开始时我们创建两个通道`answerα`和`answerβ`来接收结果，以及另一个通道`done`来跟踪计算是否完成。我们创建一个匿名函数来关闭这些通道，并使用`defer`语句来保证它们在不再需要用到时才被关闭，即外层函数返回时。接下来，我们进行昂贵的计算（分别在它们自己的`goroutine`里进行），每一个计算使用的都是独立分配的数据、独立的结果通道以及共享的`done`通道。

我们本可以让每一个计算都使用相同的`answer`通道，但如果真那样做的话我们就不知道哪个计算返回的是哪个结果了（当然这可能也没关系）。如果我们想让每个计算共享相同的通道，同时又想为不同的结果标记其源头，我们可以使用一个操作一个结构体的通道，例如，`type Answer struct{id, answer int}`。

这两个计算开始于各自的`goroutine`中（但是是阻塞的，因为它们的通道是非缓冲的）之后，我们就可以从它们那里获取结果。每次迭代

时，`for`循环中的`which`和`result`值都是全新的，而其阻塞的`select`语句会任意选择一个已准备好的`case`语句执行。如果一个结果已经准备好了，我们会设置`which`来标记它的源头，并将该源头与结果打印出来。如果`done`通道准备好了，我们将 `doneCount` 计数器加 1。当其值达到我们预设的需要计算的个数时，就表示所有计算都完成了，`for`循环结束。

一旦跳出`for`循环后，我们就知道两个进行计算的`goroutine`都不会再发送数据到通道里去（因为它们完成时会自动跳出它们自身的无限循环，参见5.4节）。当函数返回时，`defer`语句中会自动将通道关闭，而其所使用的资源也会被释放。这样，垃圾回收器就会清理这几个`goroutine`，因为它们不再需要执行，并且所使用的通道也已被关闭。

Go语言的通信和并发特性非常灵活而功能强大，第7章将专门阐述该主题。

5.5 defer、panic和recover

`defer`语句用于延迟一个函数或者方法（或者当前所创建的匿名函数）的执行，它会在外围函数或者方法返回之前但是其返回值（如果有的话）计算之后执行。这样就有可能在一个被延迟执行的函数内部修改函数的命名返回值（例如，使用赋值操作符给它们赋新值）。如果一个函数或者方法中有多个`defer`语句，它们会以LIFO（Last In First Out，后进先出）的顺序执行。

`defer`语句最常用的用法是，保证使用完一个文件后将其成功关闭，或者将一个不再使用的通道关闭，或者捕获异常。

```
var file *os.File
var err error
if file, err = os.Open(filename); err != nil {
```

```
    log.Println( " failed to open the file " , err)
    return
}
```

defer file.Close()

这段代码摘自wordfrequency程序的updateFrequencies()函数，我们在之前的章节中讨论过它。这里展示了一个典型的模式，即在打开文件并在文件打开成功后用延迟执行的方式保证将其关闭。

该模式创建了一个值，并在该值被垃圾收集之前延迟执行一些关闭函数来清理该值（例如，释放一些该值所使用的资源）。这个模式在Go语言中是一个标准做法[\[8\]](#)。虽然很少用到，我们当然也可以将该模式应用于自定义类型，为类型定义Close()或者Cleanup()方法，并将该方法用defer语法调用。

panic和recover

通过内置的panic()和recover()函数，Go语言提供了一套异常处理机制。类似于其他语言（例如，C++、Java和Python）中所提供的异常机制，这些函数也可以用于实现通用的异常处理机制，但是这样做在Go语言中是不好的风格。

Go语言将错误和异常两者区分对待。错误是指可能出错的东西，程序需以优雅的方式将其处理（例如，文件不能被打开）。而异常是指“不可能”发生的事情（例如，一个应该永远为true的条件在实际环境中却是false的）。

Go语言中处理错误的惯用法是将错误以函数或者方法最后一个返回值的形式将其返回，并总是在调用它的地方检查返回的错误值（不过通常在将值打印到终端的时候会忽略错误值。）

对于“不可能发生”的情况，我们可以调用内置的panic()函数，该函数可以传入任何想要的值（例如，一个字符串用于解释为什么那些不变的东西被破坏了）。在其他语言中，这种情况下我们可能使用一个

断言，但在Go语言中我们使用`panic()`。在早期开发以及任何发布阶段之前，最简单同时也可能是最好的方法是调用 `panic()`函数来中断程序的执行以强制发生错误，使得该错误不会被忽略因而能够被尽快修复。一旦开始部署程序时，任何情况下可能发生错误都应该尽一切可能避免中断程序。我们可以保留所有 `panic()`函数但在包中添加一个延迟执行的`recover()`调用来达到这个目的。在恢复过程中，我们可以捕捉并记录任何异常（以便这些问题保留可见），同时向调用者返回非`nil`的错误值，而调用者则会试图让程序恢复到健康状态并继续安全运行。

当内置的`panic()`函数被调用时，外围函数或者方法的执行会立即中止。然后，任何延迟执行的函数或者方法都会被调用，就像其外围函数正常返回一样。最后，调用返回到该外围函数的调用者，就像该外围调用函数或者方法调用了 `panic()`一样，因此该过程一直在调用栈中重复发生：函数停止执行，调用延迟执行函数等。当到达`main()`函数时不再有可以返回的调用者，因此这时程序会终止，并将包含传入原始`panic()`函数中的值的调用栈信息输出到`os.Stderr`。

上面所描述的只是一个异常发生时正常情况下所展开的。然而，如果其中有个延迟执行的函数或者方法包含一个对内置的`recover()`函数（可能只在一个延迟执行的函数或者方法中调用）的调用，该异常展开过程就会终止。这种情况下，我们就能够以任何我们想要的方式响应该异常。有种解决方案是忽略该异常，这样控制权就会交给包含了延迟执行的`recover()`调用的函数，该函数然后会继续正常执行。我们通常不推荐这种方法，但如果使用了，至少需要将该异常记录到日志中以不完全隐藏该问题。另一种解决方案是，我们完成必要的清理工作，然后手动调用 `panic()`函数来让该异常继续传播。一个通用的解决方案是，创建一个 `error`值，并将其设置成包含了`recover()`调用的函数的

返回值（或返回值之一），这样就可以将一个异常（即一个`panic()`）转换成错误（即一个`error`）。

绝大多数情况下，Go语言标准库使用`error`值而非异常。对于我们自己定义的包，最好别使用`panic()`。或者，如果要使用`panic()`，也要避免异常离开这个自定义包边界，可以通过使用`recover()`来捕捉异常并返回一个相应的错误值，就像标准库中所做的那样。

一个说明性的例子是Go语言中最基本的正则表达式包`regexp`。该包中有一些函数用于创建正则表达式，包括`regexp.Compile()`和`regexp.MustCompile()`。第一个函数返回一个编译好的正则表达式和`nil`，或者如果所传入的字符串不是个合法的正则表达式，则返回`nil`和一个`error`值。第二个函数返回一个编译好的正则表达式，或者在出问题抛出异常。第一个函数非常适合于当正则表达式来自于外部源时（例如，当来自于用户输入或者从文件读取时）。第二个函数非常适合于当正则表达式是硬编码在程序中时，这样可以保证如果我们不小心对正则表达式犯了个错误，程序会因为异常而立即退出。

什么时候应该允许异常终止程序，什么时候又应该使用`recover()`来捕捉异常？有两点相互冲突的利益需要考虑。作为一个程序员，如果程序中有逻辑错误，我们希望程序能够立马崩溃，以便我们可以发现并修改该问题。但一旦程序部署好了，我们就不想让我们的程序崩溃。

对于那些只需通过执行程序（例如，一个非法的正则表达式）就能够捕捉的问题，我们应该使用`panic()`（或者能够发生异常的函数，如`regexp.MustCompile()`）、因为我们永远不会部署一个一运行就崩溃的程序。我们要小心只在程序运行时一定会被调用到的函数中才这样做，例如`main`包中的`init()`函数（如果有的话）、`main`包中的`main()`函数，以及任何我们的程序所导入的自定义包中的`init()`函数，当然也包括这些函数所调用的任何函数或者方法。如果我们在使用测试套件，

我们当然可以把异常的使用扩展至测试套件会调用到的任何函数或者方法。自然地，我们必须保证无论程序的控制流程如何进行，潜在的异常的情况总是能够被适当地处理。

对于任何特殊情况下可能运行也可能不运行的函数或者方法，如果调用了`panic()`函数或者调用了发生异常的函数或者方法，我们应该使用`recover()`以保证将异常转换成错误。理想情况下，`recover()`函数应该在尽可能接近于相应`panic()`的地方被调用，并在设置其外围函数的`error`返回值之前尽可能合理的将程序恢复到健康状态。对于`main`包的`main()`函数，我们可以放入一个“捕获一切”的`recover()`函数，用于记录任何捕获的异常。但不幸的是，延迟执行的`recover()`函数被调用后程序会终止。稍后我们会看到，我们可以绕过这个问题。

接下来让我们看两个例子，第一个演示了如何将异常转换成错误，第二个例子展示了如何让程序变得更健壮。

假设我们有如下函数，它在我们所使用的某个包的深处。但我们没法更改这个包，因为它来自于一个我们无法控制的第三方。

```
func ConvertInt64ToInt(x int64) int {  
    if math.MinInt32 <= x && x <= math.MaxInt32 {  
        return int(x)  
    }  
    panic(fmt.Sprintf( " %d is out of the int32 range " , x))  
}
```

该函数安全地将一个`int64`类型的值转换成一个`int`类型的值，如果该转换产生的结果非法，则报告发生异常。

为什么一个这样的函数优先使用`panic()`呢？我们可能希望一旦有错就强制崩溃，以便尽早弄清楚程序错误。另一种使用案例是，我们有一个函数调用了一个或者多个其他函数，一旦出错我们希望尽快返回到原始调用函数，因此我们让被调用的函数碰到问题时抛出异常，

并在调用处使用`recover()`捕获该异常（无论异常来自哪里）。正常情况下，我们希望包报告错误而非抛出异常，因此常用的做法是在一个包内部使用`panic()`，同时使用`recover()`来保证产生的异常不会泄露出去，而只是报告错误。另一种使用案例是，将类似`panic(" unreachable ")`这样的调用放在一个我们从逻辑上判断不可能到达的地方（例如函数的末尾，而该函数总是会在到达末尾之前通过`return`语句返回），或者在一个前置或者后置条件被破坏时才调用`panic()`函数。这样做可以保证，如果我们破坏了函数的逻辑，立马就能够知道。

如果以上理由没有一个成立，那么当问题发生时我们就应该避免崩溃，而只是返回一个非空的`error`值。因此，在本例中，如果转换成功，我们希望返回一个`int`型值和一个`nil`，如果失败则返回一个`int`值和一个非空的错误值。下面是一个包装函数，能够实现我们想要的功能。

```
func IntFromInt64(x int64) (i int, err error){
    defer func(){
        if e := recover(); e != nil{
            err = fmt.Errorf( " %v " , e)
        }
    }()
    i = ConvertInt64ToInt(x)
    return i, nil
}
```

该函数被调用时，Go语言会自动地将其返回值设置成其对应类型的零值，如在这里是0和`nil`。如果对自定义的`ConvertInt64ToInt()`函数正常返回，我们将其值赋值给`i`返回值，并返回`i`和一个表示没错误发生的`nil`值。但是如果`ConvertInt64ToInt()`函数抛出异常，我们可以在延迟执

行的匿名函数中捕获该异常，并将err设置成一个错误值，其文本为所捕获错误的文本表示。

如IntFromInt64()函数所示，可以非常容易将异常转换成错误值。

对于我们第二个例子，我们考虑如何让一个 Web 服务器在遇到异常时仍能够健壮地运行。我们回顾下第2章中的statistics例子（参见2.4节）。如果我们在那个服务器端犯了个程序错误，例如，我们意外地传入了一个nil值作为image.Image值，并调用它的一个方法，我们可能得到一个如果不调用 recover()函数就会导致程序中止的异常。如果网站对我们来说非常重要，特别是我们希望在无人值守的情况下持续运行时，这当然是让人非常不满意的场景。我们期望的是即使出现异常服务器也能继续运行，同时将任何异常都以日志的形式记录下来，以便将我们进行跟踪并在有时间时将其修复。

我们创建了一个statistics例子的修改版（事实上，是statistics_ans解决方案的修改版），保存在文件statistics_nonstop/statistics.go中。为了测试需要，我们所做的修改是在网页上添加一个额外的“Panic!”按钮，点击后可产生一个异常。其中所做的最重要的修改是，我们让服务器可以从异常恢复。为了更好地查看发生了什么，每当成功响应一个客户端，或者当我们得到一个错误的请求时，或者如果服务器重启了，我们都以日志的形式将其记录下来。下面是一个常规日志的小样本。

```
[127.0.0.1:41373] served OK
[127.0.0.1:41373] served OK
[127.0.0.1:41373] bad request: '6y' is invalid
[127.0.0.1:41373] served OK
[127.0.0.1:41373] caught panic: user clicked panic button!
[127.0.0.1:41373] served OK
```

为了让输出结果更适合于阅读，我们告诉log包不要打印时间戳。

在了解我们对代码做了什么更改之前，让我们简单地回顾下原始代码。

```
func main(){
    http.HandleFunc( "/" , homePage)
    if err := http.ListenAndServe( ":9001 " , nil); err != nil {
        log.Fatal( " failed to start server " , err)
    }
}

func homePage(writer http.ResponseWriter, request *http.Request) {
    // ...
}
```

虽然我们所要展示的技术可应用于创建有多个网页的网站，但这里这个网站只有一个网页。如果发生了异常而没有被`recover()`捕获，即该异常被传播到了`main()`函数，服务器就会终止，这就是我们所要阻止的。

```
func homePage(writer http.ResponseWriter, request *http.Request) {
    defer func() { // 每一个页面都需要
        if x := recover(); x != nil {
            log.Printf( " [%v] caught panic: %v " , request.RemoteAddr, x)
        }
    }()
    // ...
}
```

对于能够健壮地应对异常的Web服务器而言，我们必须保证每一个页面响应函数都有一个调用 `recover()`的匿名函数。这可以阻止异常的蔓延。然而，这不会阻止页面响应函数返回（因为延迟执行的语句只

是在函数的返回语句之前执行），但这不重要，因为每次页面被请求时，`http.ListenAndServe()`函数会重新调用页面响应函数。

当然，对于一个含有大量页面处理函数的网站，添加一个延迟执行的函数来捕获和记录异常会产生大量重复的代码，并且容易被遗漏。我们可以通过将每个页面处理函数都需要的代码包装为一个函数来解决这个问题。使用包装函数，只要改变下`http.HandleFunc()`函数的调用，我们可以从页面处理函数中移除恢复代码。

```
http.HandleFunc( " / " , logPanics(homePage))
```

这里我们使用原始的`homePage()`函数（即未调用延迟执行`recover()`的版本），它依赖于`logPanics()`包装函数来处理异常。

```
func logPanics(function func(http.ResponseWriter,  
    *http.Request)) func(http.ResponseWriter, *http.Request) {  
    return func(writer http.ResponseWriter, request *http.Request) {  
        defer func() {  
            if x := recover(); x != nil {  
                log.Printf( " [%v] caught panic: %v " , request.RemoteAddr,  
                    x)  
            }  
        }()  
        function(writer, request)  
    }  
}
```

该函数接收一个 HTTP 处理函数作为其唯一参数，创建并返回一个匿名函数。该匿名函数包含一个延迟执行的（同时也是）匿名函数以捕获并记录异常，然后调用所传入的处理函数。这跟我们在上面修改过的`homePage()`函数中所看到的效果一样，它添加了一个延迟执行的异常捕获器和日志记录器，但是更为方便，因为我们无需为每一个页

面处理函数添加一个延迟执行函数。相反，我们使用`logPanic()`包装器将每个页面处理函数传入`http.HandleFunc()`。

文件`statistics_nonstop2/statistics.go`中有使用该技术的`statistics`程序的版本。匿名函数的内容将在下一节中关于闭包的节中详细阐述（参见5.6.3节）。

5.6 自定义函数

函数是面向过程编程的根本，Go语言原生支持函数。Go语言的方法（在第6章描述）和函数是很相似的，所以本章的主题和过程编程以及面向对象编程都相关。下面是函数定义的基本语法。

```
func functionName(optionalParameters) optionalReturnType {  
    body  
}  
  
func functionName(optionalParameters) (optionalReturnValues) {  
    body  
}
```

函数可以有任意多个参数，如果没有参数那么圆括号是空的，否则要写成这样：`params1 type1,..., paramsN typeN`，其中`params1`是参数，`type1`是参数类型，多个参数之间要用逗号分隔开。参数必须按照给定的顺序来传递，没有和Python的命名参数相同的功能。不过Go语言里也可以实现一种类似的效果，后面就可以看到（5.6.1.3节）。

如果来实现可变参数，可以将最后一个参数的类型之前写上省略号，也就是说，函数可以接收任意多个那个类型的值，在函数里，实际上这个参数的类型是`[]type`。

函数的返回值也可以是任意个，如果没有，那么返回值列表的右括号后面是紧接着左大括号的。如果只有一个返回值可以直接写返回的类型，如果有两个或者多个没有命名的返回值，必须使用括号而且得这样写(`type1,..., typeN`)。如果有一个或者多个命名的返回值，也必须使用括号，要写成这样(`values1 type1,..., valuesN typeN`)，其中`values1`是一个返回值的名称，多个返回值之间必须使用逗号分隔开。函数的返回值可以全部命名或者全都不命名，但不能只是部分命名的。

如果函数有返回值，则函数必须至少有一个`return`语句或者最后执行`panic()`调用。如果返回值不是命名的，则`return`语句必须指定和返回值列表一样多的值。如果返回值是命名的，则`return`语句可以像没有命名的返回值方式一样或者是一个空的`return`语句。注意尽管空的`return`语句是合法的，但它被认为是一种拙劣的写法，我们这本书所有的例子都没有这样写。

如果函数有返回值，则函数的最后一个语句必须是一个`return`语句或者`panic()`调用。如果函数是以抛出异常结束，Go 编译器会认为这个函数不需要正常返回，所以也就不需要这个`return`语句。但是如果函数是以`if`语句或`switch`语句结束，且这个`if`语句的`else`分支以`return`语句结尾或者`switch`语句的`default`分支以`return`语句结尾的话，Go编译器还无法意识到它们后面已经不需要`return`语句。对于这种情况的解决方法有几种，要么不给`if`语句和`switch`语句添加对应的`else`语句和`default`分支，要么将`return`语句放到`if`或者`switch`后面，或者在最后简单地加上一句`panic(" unreachable ")`语句，我们前面看到过这种做法（5.2.2.1节）。

5.6.1 函数参数

我们之前见过的函数都是固定参数和指定类型的，但是如果参数的类型是`interface{}`，我们就可以传递任何类型的数据。通过使用接口

类型参数（无论是自定义接口类型还是标准库里定义的接口类型），我们可以让所创建的函数接受任何实现特定方法集合的类型作为参数，我们在6.3节会继续讨论这个问题。

这一节我们来了解关于函数参数的其他内容。第一个小节关于如何将函数的返回值作为其他函数的参数，第二小节讨论可变参数，最后我们讨论如何实现可选参数。

5.6.1.1 将函数调用作为函数的参数

如果我们有一个函数或者方法，接收一个或者多个参数，我们可以理所当然地直接调用它并给它相应的参数。另外，我们可以将其他函数或者方法调用作为一个函数的参数，只要该作为参数的函数或者方法的返回值个数和类型与调用函数的参数列表匹配即可。

下面是一个例子，一个函数要求传入三角形的边长（以 3 个整型数的方式），然后使用海伦公式计算出三角形的面积。

```
for i := 1; i <= 4; i++ {  
    a, b, c := PythagoreanTriple(i, i+1)  
    Δ1 := Heron(a, b, c)  
    Δ2 := Heron(PythagoreanTriple(i, i+1))  
    fmt.Printf( " Δ1 == %10f == Δ2 == %10fn ", Δ1, Δ2)  
}
```

```
Δ1 == 6.000000 == Δ2 == 6.000000  
Δ1 == 30.000000 == Δ2 == 30.000000  
Δ1 == 84.000000 == Δ2 == 84.000000  
Δ1 == 180.000000 == Δ2 == 180.000000
```

首先我们使用欧几里德的勾股函数来获得边长，然后将这3个边长作为Heron()的参数，应用海伦公式来计算面积。我们重复一次这个计算过程，不过这次我们是直接将PythagoreanTriple()函数作为Heron()函

数的参数，交由Go语言将PythagoreanTriple()函数的3个返回值转换成Heron()函数的参数。

```
func Heron(a, b, c int) float64 {  
     $\alpha$ ,  $\beta$ ,  $\gamma$  := float64(a), float64(b), float64(c)  
    s := ( $\alpha$  +  $\beta$  +  $\gamma$ ) / 2  
    return math.Sqrt(s * (s -  $\alpha$ ) * (s -  $\beta$ ) * (s -  $\gamma$ ))  
}  
  
func PythagoreanTriple(m, n int) (a, b, c int) {  
    if m < n {  
        m, n = n, m  
    }  
    return (m * m) - (n * n), (2 * m * n), (m * m) + (n * n)  
}
```

为了阅读完整性，这里给出了Heron()和PythagoreanTriple()函数的实现。这里PythagoreanTriple()函数使用了命名返回值（算是对该函数文档的一些补充）。

5.6.1.2 可变参数函数

所谓可变参数函数就是指函数的最后一个参数可以接受任意个参数。这类函数在最后一个参数的类型前面添加有一个省略号。在函数里面这个参数实质上变成了一个对应参数类型的切片。例如，我们有一个签名是Join(xs...string)的函数，xs的类型其实是[]string。

下面是一个使用可变参数的例子，它返回输入的整数里最小的一个。我们将分析它的调用过程以及输出的结果。

```
fmt.Println(MinimumInt1(5, 3), MinimumInt1(7, 3, -2, 4, 0, -8, -5))
```

`MinimumInt1()`函数可以传入一个或者多个整型数，然后返回其中最小的一个。

```
func MinimumInt1(first int, rest...int) int {  
    for _, x := range rest {  
        if x < first {  
            first = x  
        }  
    }  
    return first  
}
```

我们可以很容易地实现一个任意参数（即使不传参数也可以）的函数，例如`MinimumInt0 (ints...int)`，或者至少是两个整型数的函数，例如，`MinimumInt2(first, second, int, rest...int)`。

假如我们有一个`[]int`类型的切片，我们可以这样使用`MinimumInt1()`函数。

```
numbers := []int{7, 6, 2, -1, 7, -3, 9}  
fmt.Println(MinimumInt1(numbers[0], numbers[1:]...))
```

-3

函数`MinimumInt1()`至少需要一个`int`型的参数，当调用一个可变参数函数或者方法时，我们可以在一个`slice`后面放一个省略号，这样就把切片变成了一系列参数，每个参数对应切片里的一项。（我们之前在4.2.3节讨论Go语言内置的`append()`函数时讨论过。）所以我们这里实际上就是将`numbers[1:]...`展开成独立的每一个参数`6,-2,-1,7,-3,9`了，而这些都是会被保存在`rest`这个切片里面。如果我们使用刚才提到过的`MinimumInt0()`函数，我们简单地调用`MinimumInt0(numbers...)`即可。

5.6.1.3 可选参数的函数

Go语言并没有直接支持可选参数。但是，要实现它也不难，只需增加一个额外的结构体即可，而且Go语言能保证所有的值都会被初始化为零值。

假设我们有一个函数用来处理一些自定义的数据，默认就是简单地处理所有的数据，但有些时候我们希望可以指定处理第一个或者最后一个项，还有是否记录函数的行为，或者对于非法的项做错误处理，等等。

一个办法就是创建一个签名为 `ProcessItems(items Items, first, last int, audit bool, errorHandler func(item Item))` 的函数。在这个设计里，如果 `last` 的值为0的话意味着需要取到最后一个item而不用管这个索引值，而 `errorHandler` 函数只有在不为 `nil` 时才会被调用。也就是说，不管在哪调用它，如果希望是默认行为的话，只需要写 `ProcessItems(items, 0, 0, false, nil)` 就可以了。

一个比较优雅的做法就是这样定义函数 `ProcessItems(items Items, options Options)`，其中 `Options` 结构体保存了所有其他参数的值，初始值均为零值。这样大部分调用都可以被简化为 `ProcessItems(items, Options{})`。然后在我们需要指定一个或者多个额外参数的场合，我们可以为 `Options` 结构指定一到多个字段的值（我们会在 6.4 节详细描述结构体）。让我们来看看如何用代码实现，先从 `Options` 结构开始。

```
type Options struct {  
    First    int    // 要处理的第一项  
    Last     int    // 要处理的最后一项（0意味着要从第一项  
    开始处理所有项）  
    Audit    bool   // 如果为true,所有动作都被记录  
    ErrorHandler func(item Item) // 如果不是nil，对每一个坏项周  
    用一次  
}
```

一个结构体能够聚合或者嵌入一个或者多个任何类型的字段（关于聚合和嵌入的区别将在第6章详细描述）。这里，`Options`结构体聚合了两个`int`型字段、一个`bool`型字段以及一个签名为`func(Item)`的函数，其中`Item`是某自定义类型。

```
ProcessItems(items, Options{})
```

```
errorHandler := func(item Item) { log.Println( " Invalid: " , item) }
```

```
ProcessItems(items, Options{Audit: true, ErrorHandler: errorHandler})
```

这块代码调用了两次自定义函数`ProcessItems()`，第一次调用使用默认的选项（例如，处理所有的项，但是不记录任何的动作，对于非法的记录也不调用错误处理函数来处理），第二次调用时创建了一个`Options`值，其中`Options`的`First`字段和`Last`字段是0（也就是告诉这个函数要处理所有的项），但设置了`Audit`和`ErrorHandler`字段这样函数就能记录它的行为而且当发现非法的项时能够做一些相应的处理。

这种利用结构体来传递可选参数的技术在标准库里也有用到，例如，`image.jpeg.Encode()`函数，我们在后面的6.5.2节还会看到这种技术。

[5.6.2 init\(\)函数和main\(\)函数](#)

Go语言为特定目的保留了两个函数名：`init()`函数（可以出现在任何的包里）和`main()`函数（只在`main`包里）。这两个函数既不可接收任何参数，也不返回任何结果，一个包里可以有很多`init()`函数。但是我写这本书的时候，Go编译器只支持每个包最多一个`init()`函数，所以我们推荐你在一个包里最多只用一个`init()`函数。

`init()`函数和`main()`函数是自动执行的，所以我们不应该显式调用它们。对程序或者包来说`init()`是可选的，但是每一个程序必须在`main`包里包含一个`main()`函数。

Go程序的初始化和执行总是从main包开始，如果main包里导入了其他的包，则会按顺序将它们包含进 main 包里。如果一个包被其他的包多次导入的话，这个包实际上只会被导入一次（例如，有好些包都会导入 fmt 这个包，一旦导入之后再遇到就不会再次导入）。当一个包被导入时，如果它自己还导入了其他的包，则还是先将其他的包导入进来，然后再创建这个包的一些常量和变量。再接着就是调用init()函数了（如果有多个就调用多次），最终所有的包都会被导入到main包里（包括这些包所导入的包等），这时候main这个包的常量和变量也会被创建，init()函数会被执行（如果有或者多个的话）。最后，main包里的main()函数会被执行，程序开始运行。这些事件的过程如图5-1所示。

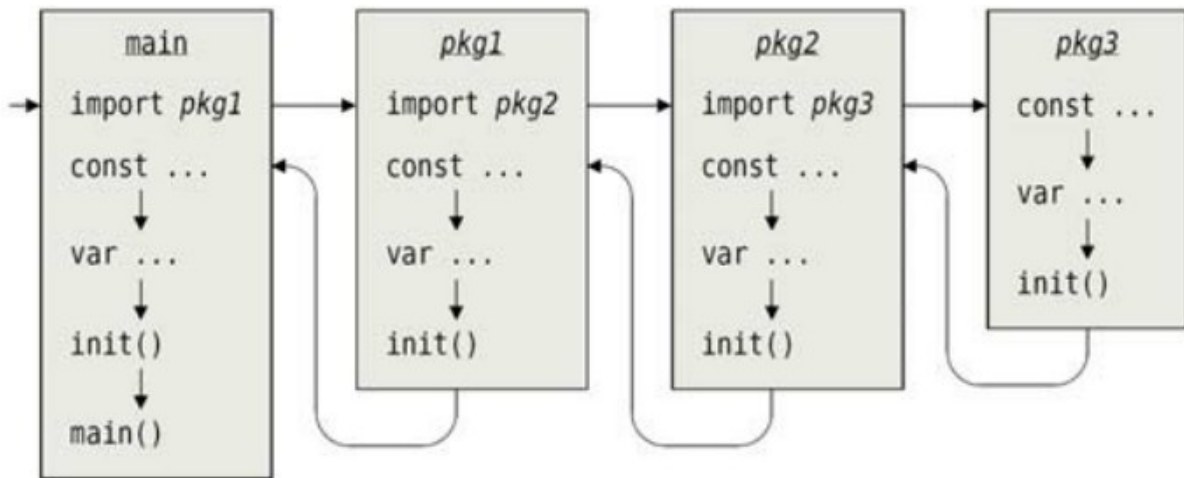


图5-1 程序的启动顺序

我们可以在init()函数里写一些go语句，但是要注意的是init()函数会在main()函数之前执行，所以init()中不应该依赖任何在main()函数里创建的东西。

让我们来看一个例子（从第1章的americanise/americanise.go文件里截取），看看实际会发生什么事情。

```
package main
```

```

import (
    "bufio "
    "fmt "
    //...
    "strings "
)
var britishAmerican = " british-american.txt "
func init() {
    dir, _ := filepath.Split(os.Args[0])
    britishAmerican = filepath.Join(dir, britishAmerican)
}
func main() {
    //...
}

```

Go程序从main包开始，因为main包里导入了其他的包，所以它先按顺序从bufio包开始把其他的包导进来。bufio包自身也导入了一些其他的包，所以这些导入会先完成。在导入每一个包时总是先去将这个包的所有依赖包导入，然后才创建包级别的常量和变量，再接着执行这个包的init()函数。bufio包导入完成后fmt包会被导入。fmt包里它自己也导入了strings包，所以当Go语言会忽略main包导入strings包的语句，因为strings包之前已被导入。

当所有的包被导入后，包级别的britishAmerican变量会被创建，然后main包里的init()函数会被调用。最后main()函数被调用，程序开始执行。

5.6.3 闭包

所谓闭包就是一个函数“捕获”了和它在同一作用域的其他常量和变量。这就意味着当闭包被调用的时候，不管在程序什么地方调用，闭包能够使用这些常量或者变量。它不关心这些捕获了的变量和常量是否已经超出了作用域，所以只要闭包还在使用它，这些变量就还会存在。

在Go语言里，所有的匿名函数（Go语言规范中称之为函数字面量）都是闭包。

闭包的创建方式和普通函数在语法上几乎一致，但有一个关键的区别：闭包没有名字（所以func关键字后面紧接着左括号）。通常都是通过将闭包赋值给一个变量来使用闭包，或者将它放到一个数据结构里（如映射或者切片）。

我们已经见过好几个闭包的例子，例如，当我们使用defer语句或者匿名函数的时候。我们在这本书的一些例子里也创建过闭包，如americanise例子里使用的makeReplacerFunction()函数（1.6节），在第3章中当我们将匿名函数作为参数传递给strings.FieldsFunc()或者strings.Map()函数时（3.6.1节），还有这一章之前的createCounter()函数和logPanics()函数。不过我们在这里还会介绍一些简单的例子。

闭包的一种用法就是利用包装函数来为被包装的函数预定义一到多个参数。例如，假如我们想给大量文件增加不同后缀，本质上就是要包装string的+连接操作符，一个参数会不断变化（文件名）而另一个参数为固定值（后缀名）。

```
addPng := func(name string) string { return name + ".png" }
addJpg := func(name string) string { return name + ".jpg" }
fmt.Println(addPng("filename"), addJpg("filename"))
```

```
filename.png filename.jpg
```


`addPng`和`addJpg`变量都是对匿名函数（即闭包）的引用。这种引用可以像上述代码段中说明的那样像正常命名的函数那样被调用。

现实环境中当我们需要创建很多类似的函数时，相比一个个单独创建，我们经常会用到一个工厂函数（**factory function**），工厂函数返回一个函数。下面就是一个工厂函数的例子。它返回一个函数。如果接收到的文件名不带后缀名，那么这个函数就为它增加一个后缀名。

```
func MakeAddSuffix(suffix string) func(string) string {  
    return func(name string) string {  
        if !strings.HasSuffix(name, suffix) {  
            return name + suffix  
        }  
        return name  
    }  
}
```

工厂函数`MakeAddSuffix()`返回的闭包在创建时捕获了`suffix`变量。这个返回的闭包接收一个字符串参数（如文件名）并返回添加了被捕获的`suffix`的文件名。

```
addZip := MakeAddSuffix( ".zip " )  
addTgz := MakeAddSuffix( ".tar.gz " )  
fmt.Println(addTgz( " filename " ), addZip( " filename " ), addZip( "  
gobook.zip " ))
```

```
filename.tar.gz filename.zip gobook.zip
```

这里创建了两个闭包`addZip()`和`addTgz()`并调用了它们。

[5.6.4 递归函数](#)

递归函数就是调用自己的函数，还有相互递归函数就是相互调用对方的函数。Go语言完全支持递归函数。

递归函数通常有相同的结构：一个跳出条件和一个递归体，所谓跳出条件就是一个条件语句，例如 `if` 语句等，根据传入的参数判断是否需要停止递归，而递归体则是函数自身所做的一些处理，包括最少也得调用自身一次（或者调用它相互递归的另一个函数），而且递归调用时所传入的参数一定不能和当前函数传入的一样，在跳出条件里还会检查是否可以结束递归。

递归函数非常便于实现递归的数据结构例如二叉树，但是对于数值计算而言可能会性能比较低下。

我们从一个非常简单（性能也比较低）的示例开始介绍一下如何实现递归。首先我们看一个对递归函数的调用和相应输出，然后我们再看看递归函数本身。

```
for n := 0; n < 20; n++ {  
    fmt.Print(Fibonacci(n), " ")  
}  
fmt.Println()
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

`Fibonacci()`函数返回一个包含`n`个数字的斐波那契数列。

```
func Fibonacci(n int) int {  
    if n < 2 {  
        return n  
    }  
    return Fibonacci(n-1) + Fibonacci(n-2)  
}
```

上面的if 语句是这个递归函数的跳出条件，用它来保证递归最终是可以结束的。这是因为不管我们当初指定的n是什么，每一次递归调用函数自身的时候，传给递归函数的n值都会减少，因此n的值在某个时刻必然会小于2。

举个例子，如果我们调用Fibonacci(4)，跳出条件不会被触发，函数返回两个递归调用Fibonacci(3)和Fibonacci(2)的和，前者会递归调用Fibonacci(2)(同理，Fibonacci(2)也会调用Fibonacci(1)和Fibonacci(0))和Fibonacci(1)，后者则会递归调用Fibonacci(1)和Fibonacci(0)，一旦n小于2则递归就会返回，这个过程可以用图5-2来描述。

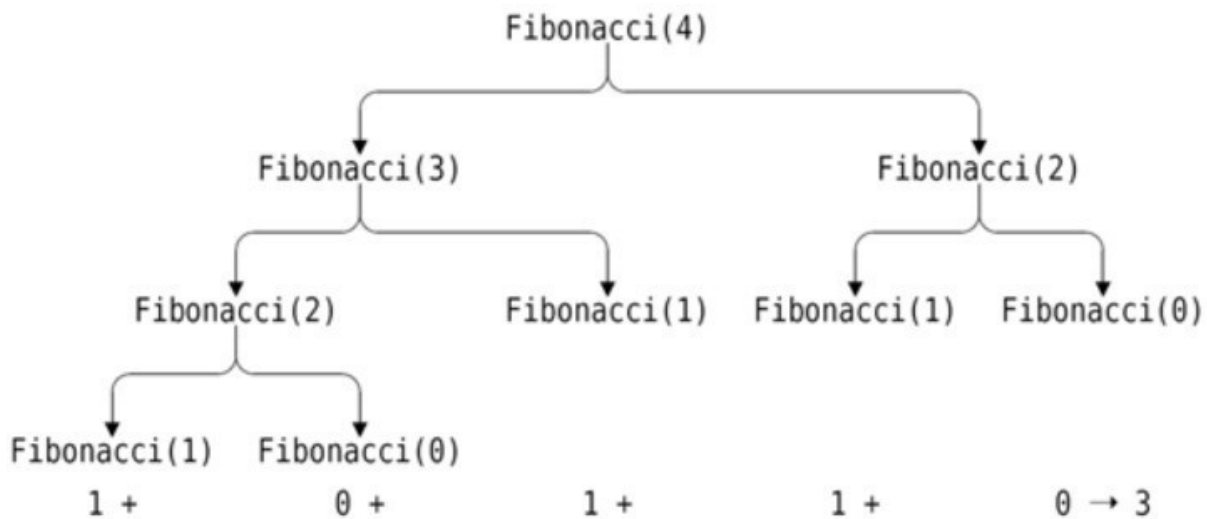


图5-2 递归的Fibonacci

显然，Fibonacci()做了很多重复的计算，尽管我们只是输入了一个很小的数如4。我们后面会看到如何避免这个问题（参见5.6.7节）。

Hofstadter 男女序列就是一个使用相互递归函数的例子，下面的代码将每个序列中的前 20个整数打印出来：

```
females := make([int, 20)
males := make([int, len(females))
for n := range females {
```

```

    females[n] = HofstadterFemale(n)
    males[n] = HofstadterMale(n)
}
fmt.Println( " F " , females)
fmt.Println( " M " , males)
F [1 1 2 2 3 3 4 5 5 6 6 7 8 8 9 9 10 11 11 12]
M [0 0 1 2 2 3 4 4 5 6 6 7 7 8 9 9 10 11 11 12]

```

下面是产生以上序列的两个相互递归函数。

```

func HofstadterFemale(n int) int {
    if n <= 0 {
        return 1
    }
    return n - HofstadterMale(HofstadterFemale(n-1))
}

func HofstadterMale(n int) int {
    if n <= 0 {
        return 0
    }
    return n - HofstadterFemale(HofstadterMale(n-1))
}

```

通常在函数的开始处都会有一个跳出条件用来确保递归能够正常结束，在递归发生的地方我们递归传入的参数是一个不断减少的值，最终跳出条件会被满足。

其他语言实现的Hofstadter函数通常会有一个问题，那就是HofstadterFemale()函数是在HofstadterMale()之前定义的，但是HofstadterFemale()却调用了HofstadterMale()函数。这些编程语言将要求

我们预声明HofstadterMale()函数。Go语言没有这样的限制，因为Go语言允许函数以任何顺序定义。

我们再来看最后一个递归的例子，它判断一个单词是否是一个回文单词（也就是单词反转后和原单词是一模一样的，如“PULLUP”和“ROTOR”都是回文单词）。

```
func IsPalindrome(word string) bool {  
    if utf8.RuneCountInString(word) <= 1 {  
        return true  
    }  
    first, sizeOfFirst := utf8.DecodeRuneInString(word)  
    last, sizeOfLast := utf8.DecodeLastRuneInString(word)  
    if first != last {  
        return false  
    }  
    return IsPalindrome(word[sizeOfFirst : len(word)-sizeOfLast])  
}
```

函数第一部分是一个跳出条件：如果单词的长度为0或者1，那么就认为这是一个回文，所以直接返回 true。而函数体所用的算法，是比较首字母和尾字母，如果它们不同，那么这个单词肯定不是回文，我们可以立即返回false。但是如果首字符和尾字符是相同的，那我们就递归判断这个单词的一个字串（去掉首尾两个字符）是否是回文。

举个例子，我们传入一个字符串“PULLUP”，函数首先比较首字符“P”和尾字符“P”，然后它调用自己判断子字符串“ULLU”，再比较“U”和“U”，然后再递归调用判断子字符串“LL”，比较“L”和“L”，最后它递归调用时传的是一个空的字符串。同样，对于“ROTOR”这个字符串，首先函数是比较首字符“R”和尾字符“R”，然后递归判断“OTO”，比较首字符“O”和尾字符“O”，最后递归判断一个单字符“T”。这两种情况，

函数都返回 `true`。但对“DECIDED”字符串，函数先比较“D”和“D”，然后递归判断“ECIDE”，比较“E”和“E”，当判断到“C”和“D”的时候，它返回了 `false`。

回忆一下我们在3.6.3节讲到的`utf8.DecodeRuneInString()`函数，它返回字符串的第一个字符（`rune`类型）和它占用的字节数。`utf8.DecodeLastRuneInString()`函数也类似，作用于最后一个字符，利用这两个大小，我们可以安全地将每个字符都切割出来，因为它们不会意外将一个多字节表示的字符切割成两个。

当一个函数使用尾部递归，也就是在最后执行一句递归调用，这种情况下我们可以简单地将它转换成一个循环。使用循环的好处就是可以减少递归调用的开销，因为有限的栈空间对函数的深度递归是很有影响的，虽然由于Go语言使用了自己的内存管理机制导致这个栈空间的限制相对不严重一些。（顺便提一下，后面我们有个练习让大家将递归函数`IsPalindrome()`转换成使用循环的方式实现。）当然，有些时候递归是实现算法的最好方式，我们将在第6章介绍`omap.insert()`函数的时候看到这样的一个例子。

5.6.5 运行时选择函数

在Go语言里，函数属于第一类值（`first-class value`），也就是说，你可以将它保存到一个变量（实际上是个引用）里，这样我们就可以在运行时决定要执行哪一个函数。再者，Go语言能够创建闭包意味着我们可以在运行时创建函数，所以我们对同一个函数可以有两个或者多个不同的实现（例如使用不同的算法），在使用的时候创建它们其中的一个就行。我们在下一节讨论这两种方法。

5.6.5.1 使用映射和函数引用来制造分支

在 5.2.1 节和 5.2.2.1 节中我们看过了 `ArchiveFileList()` 的所有代码，它就是根据文件的后缀名然后调用对应的函数。这个函数的版本 1 首先用的是一个 `if` 语句，总共 7 行代码，最简洁的那个版本用的是 `switch` 语句，5 行代码。但万一我们需要处理的文件后缀名多了怎么办，如果是 `if` 的话我们需要为每个额外的 `else if` 分支增加两行代码，是 `switch` 的话我们需要为每个额外的情况增加一行代码（或者两行，使用 `gofmt` 格式化分支代码的话）。如果这个函数用于文件管理的话，它很可能需要处理几百种文件后缀，从而导致这个函数就非常长。

```
var FunctionForSuffix = map[string]func(string) ([]string, error){
    ".gz " : GzipFileList, ".tar " : TarFileList, ".tar.gz " :
    TarFileList,
    ".tgz " : TarFileList, ".zip " : ZipFileList}
func ArchiveFileListMap(file string) ([]string, error) {
    if function, ok := FunctionForSuffix[Suffix(file)]; ok {
        return function(file)
    }
    return nil, errors.New( " unrecognized archive " )
}
```

现在这个版本的 `ArchiveFileList` 函数使用了映射，这个映射的键是字符串（文件后缀），值则是签名为 `func(string) ([]string, error)` 的函数（所有自定义函数 `GzipFileList()`、`TarFileList()` 和 `ZipFileList()` 都是这种类型）。

这个函数使用 `[]` 索引操作符根据给定的前缀从 `FunctionForSuffix` 结构里得到对应的函数，如果这个前缀存在则 `ok` 的值为 `true`，否则为 `false`。如果存在匹配的函数的话，执行这个函数并将文件名作为参数传递给它，返回它的结果。

这个函数比使用if或者switch语句的更具有扩展性，不管有多少个文件的前缀处理函数在FunctionForSuffix里，这个函数都可以保持不变。这不像一个很大的if或者switch语句，不但事情变得条理清晰，还可以动态地往映射里增加其他的项，而且映射查询的速度不会随着项的增加而降低 [9] 。

5.6.5.2 动态函数的创建

在运行时动态地选择函数的另一个场景便是，当我们有两个或者更多的函数实现了相同的功能时，比如使用了不同的算法等，我们不希望在程序编译时静态绑定到其中任一个函数（例如允许我们动态地选择它们来做性能测试或回归测试）。

举个例子，如果我们使用一个 7 位的ASCII 字符，我们可以写一个更加简单的IsPalindrome()函数，而在运行时动态地创建一个我们所需要的版本。

一种做法就是声明一个和这个函数签名相同的包级别的变量，然后创建一个appropriate()函数和一个init()函数。

```
var IsPalindrome func(string) bool // 保存到函数的引用
func init() {
    if len(os.Args) > 1 && (os.Args[1] == "-a" || os.Args[1] == "--ascii") {
        os.Args = append(os.Args[:1], os.Args[2:]...) // 去掉参数
        IsPalindrome = func(s string) bool { // 简单的ASCII版本
            if len(s) <= 1 {
                return true
            }
            if s[0] != s[len(s)-1] {
                return false
            }
        }
    }
}
```



```

        return IsPalindrome(s[1 : len(s)-1])
    }
} else {
    IsPalindrome = func(s string) bool { // UTF-8版本
        // ..... 同前.....
    }
}
}

```

我们根据命令行选项来决定 `IsPalindrome()` 的实现方式。如果指定了“-a”或者“--ascii”参数，我们将它从`os.Args`切片里移除（这样其他代码不需要知道和关心这个参数），然后创建一个作用于ASCII码的`IsPalindrome()`函数。这个移除的过程有些隐晦，我们的是将`os.Args`的第一个参数和第三个之后的参数组合成一个新的`os.Args`，还有我们不能在`append()`函数里使用`os.Args[0]`，因为`append()`的第一个参数必须是一个切片，所以我们用了`os.Args[:1]`，这个切片只有一个项，那就是`os.Args[0]`（参见4.2.1节）。

如果ASCII选项没有出现，我们就创建一个和之前一样的函数，既能处理ASCII编码的字符串也能处理UTF-8编码的字符串。程序其他部分`IsPalindrome()`函数可以正常地被调用，但是实际上什么代码会被执行完全取决于我们创建的是哪个版本的函数。（这个例子的源码在[palindrome/palindrome.go](#)里。）

5.6.6 泛型函数

这一章前面的部分我们创建过一个函数，找出输入整数里最小的一个并返回。同样，我们可以将这个函数应用到不同的数据类型上，甚至是字符串，只要这个类型的值支持<操作符就行。在C++里我们会

习惯创建一个泛型函数，根据类型来确定参数，这样就可以让编译器按我们的需要来创建多个版本的函数（例如，每种类型一个函数）。在我写这本书的时候，Go语言还不支持类型参数化，所以我们不得不为每一种类型都实现一个函数，如 `MinimumInt()`、`MinimumFloat()`、`MinimumString()`等。这导致对于每个类型都得有一个对应的函数（和C++一样，只不过在Go语言里每个函数必须有唯一的函数名）。

为了提高运行时的效率，Go语言提供了多种替代方法来避免创建一些除了处理的数据类型不同外其他完全相同的函数。对于那些不经常使用或者速度已经足够快的小函数而言，这些替代方法会非常便利。

下面就是一个支持泛型的`Minimum()`函数的例子。

```
i := Minimum(4, 3, 8, 2, 9).(int)
fmt.Printf( " %T %v\n " , i, i)
f := Minimum(9.4, -5.4, 3.8, 17.0, -3.1, 0.0).(float64)
fmt.Printf( " %T %v\n " , f, f)
s := Minimum( " K " , " X " , " B " , " C " , " CC " , " CA " , " D
" , " M " ).(string)
fmt.Printf( " %T %q\n " , s, s)

int 2
float64 -5.4
string " B "
```

这个函数返回一个`interface{}`类型的值，我们使用一个非检查类型断言（5.1.2节）将这个值转换成我们所期望的值。

```
unc Minimum(first interface{}, rest...interface{}) interface{} {
    minimum := first
    for _, x := range rest {
        switch x := x.(type) {
```

```

        case int:
            if x < minimum.(int) {
                minimum = x
            }
        case float64:
            if x < minimum.(float64) {
                minimum = x
            }
        case string:
            if x < minimum.(string) {
                minimum = x
            }
    }
    return minimum
}

```

该函数接受至少一个值（**first**），以及零到多个其他值（**rest**）。我们使用**interface{}**作为参数的类型，这样我们可以传入任意类型的数据。首先我们假设第一个值是最小的，然后遍历其他的参数，若发现有比当前最小值更小的，就把它设为当前最小值，最后返回**minimum**，它也是 **interface{}**类型的，所以我们需要在调用端用非检查类型断言来将它转换成一个内置数据类型。

但这段代码仍有很多地方是重复的，如if语句里的每一个**case**分支，但如果有太多重复的代码我们可以简单地在每一个**case**分支上设置一个布尔类型（例如，**change = true**），然后在**switch**语句后面增加一个if **change**语句，用来包含所有公用的代码。

很明显，使用 `Minimum()` 函数比那些类型特定的最小函数损失了一点效率，但是这种技术非常值得了解，因为在只需定义一次函数的好处抵得过类型测试的损耗和转换的不便利性时它会变得很有价值。

还有一个比较头疼的问题，上面的泛型函数处理不了实际类型为切片的 `interface{}` 参数。举个例子，下面的函数传入一个切片和与切片的项类型相同的值，返回这个值在切片里第一次出现的索引，如果不存在就返回 `-1`。

```
func Index(xs interface{}, x interface{}) int {
    switch slice := xs.(type) {
    case []int:
        for i, y := range slice {
            if y == x.(int) {
                return i
            }
        }
    case []string:
        for i, y := range slice {
            if y == x.(string) {
                return i
            }
        }
    }
    return -1
}
```

下面是一个使用 `Index()` 函数的例子及其输出结果（源代码在 `contains/contains.go` 测试程序里）。

```
xs := []int{2, 4, 6, 8}
```

```

fmt.Println( " 5 @ " , Index(xs, 5), " 6 @ " , Index(xs, 6))
ys := []string{ " C " , " B " , " K " , " A " }
fmt.Println( " Z @ " , Index(ys, " Z " ), " A @ " , Index(ys, " A " ))
5 @ -1 6 @ 2
Z @ -1 A @ 3

```

我们真正要做的只是希望能够通用的方式对待切片。我们可以仅用一个循环然后在里面用特定类型（**type-specific**）测试呢？下面的 `IndexReflectX()` 函数就是为了这个目的创建。如果我们将上述代码片段中的 `Index()` 调用替换为 `IndexReflectX()` 调用，这段代码将输出相同的结果。

```

func IndexReflectX(xs interface{}, x interface{}) int { // 啰唆的方法
    if slice := reflect.ValueOf(xs); slice.Kind() == reflect.Slice {
        for i := 0; i < slice.Len(); i++ {
            switch y := slice.Index(i).Interface().(type) {
            case int:
                if y == x.(int) {
                    return i
                }
            case string:
                if y == x.(string) {
                    return i
                }
            }
        }
    }
    return -1
}

```

这个函数用到Go语言的反射功能（由reflect包提供，9.4.9节），将xs interface{}转换成一个切片类型的reflect.Value。我们可以遍历这个切片得到我们所关心的项。这里我们轮流访问每一个项，使用reflect.Value.Interface()函数将它的值以interface{}类型提取出来，然后马上在switch里赋值给y。这就确保了y和切片里的项具有相同的类型（例如，int或者string），后面就可以直接和非检查类型断言的x值进行比较。

实际上，reflect包可以做的事情比这多得多，显然我们可以这样简化一下这个函数。

```
func IndexReflect(xs interface{}, x interface{}) int {
    if slice := reflect.ValueOf(xs); slice.Kind() == reflect.Slice {
        for i := 0; i < slice.Len(); i++ {
            if reflect.DeepEqual(x, slice.Index(i)) {
                return i
            }
        }
    }
    return -1
}
```

这里我们是使用 reflect.DeepEqual()函数来做比较的，这个函数的功能非常强大，还可以用来比较数组、切片和结构体。

下面是一个特定类型的函数，在一个切片里查找某一项的索引。

```
func IntSliceIndex(xs []int, x int) int {
    for i, y := range xs {
        if x == y {
            return i
        }
    }
}
```

```

    }
    return -1
}

```

相比泛型函数，这种写法是很简洁的，但是如果我们想增加一种类型，就不得不创建一个额外的函数，而这个函数仅仅是函数名和参数类型不同罢了。

我们可以通过使用自定义类型将泛型函数的好处（仅需实现一次算法）和类型特定函数的简便性和高效率结合在一起。下一章我们会详细介绍这种技术。

下面两个函数都是在一个切片里查找特定项的索引，其中一个特定类型的实现，另一个是泛型实现。

```

func IntIndexSlicer(ints []int, x int) int {
    return IndexSlicer(IntSlice(ints), x)
}

func IndexSlicer(slice Slicer, x interface{}) int {
    for i := 0; i < slice.Len(); i++ {
        if slice.EqualTo(i, x) {
            return i
        }
    }
    return -1
}

```

`IntIndexSlicer()`函数传入一个`[]int`型的切片和一个`int`型的整数，然后将它们传给泛型函数`IndexSlicer()`。`IndexSlicer()`操作一个`Slicer`类型的值。`Slicer`是一个自定义的接口。任何类型可以通过实现 `Slicer` 方法（`Slicer.EqualTo()`和`Slicer.Len()`）来实现此接口。

```

type Slicer interface {

```

```

    EqualTo(i int, x interface{ }) bool
    Len() int
}
type IntSlice []int
func (slice IntSlice) EqualTo(i int, x interface{ }) bool {
    return slice[i] == x.(int)
}
func (slice IntSlice) Len() int { return len(slice) }

```

我们需要在泛型函数IndexSlicer()里实现Slicer接口的这两个方法。

IntSlice是[]int的别名，这也就是为什么IntIndexSlicer()函数能直接将接收到的[]int类型值直接赋给IntSlice而不需要显式转换，并且提供这两个方法以实现Slicer接口。IntSlice.EqualTo()方法需要传入一个索引和一个值，如果这个值和切片里索引处的值相等，就返回 true。Slicer 接口指定这个值是一个通用的interface{}类型而不是int，这样其他类型的切片也可以实现Slicer接口（如FloatSlice和StringSlice），所以我们将这个值转换成实际的类型。这里使用非检查类型断言是安全的，因为我们知道这个值最终来自于对 IntSliceIndex()函数的调用，而IntSliceIndex()函数的参数为int类型。

我们也可以为其他类型的切片实现Slicer接口，然后它们也可以使用IndexSlicer()函数。

```

type StringSlice []string
func (slice StringSlice) EqualTo(i int, x interface{ }) bool {
    return slice[i] == x.(string)
}
func (slice StringSlice) Len() int { return len(slice) }

```

StringSlice和IntSlice唯一不同的地方就是切片的类型（[]string和[]int）和非检查类型断言（string和int）。FloatSlice也是一样的

([]float64和float64)。

其实最后一个例子所用的技术在之前我们讨论自定义排序的时候就见过了（参见4.2.4节），用来实现标准库的sort包里的排序函数。关于自定义接口和自定义类型将会在第6章详细描述。

当我们使用切片或者映射时，通常可以创建泛型函数，这样就不用使用类型测试和类型断言。或者，将我们的泛型函数写成高阶函数，对所有特定的类型相关逻辑进行抽象，这将在下一节描述。

5.6.7 高阶函数

所谓高阶函数就是将一个或者多个其他函数作为自己的参数，并在函数体里调用它们。让我们来看一个最简单的高阶函数，但它的功能不是马上就能看得出来的。

```
func SliceIndex(limit int, predicate func(i int) bool) int {  
    for i := 0; i < limit; i++ {  
        if predicate(i) {  
            return i  
        }  
    }  
    return -1  
}
```

这个函数很普通，返回 predicate()为真时的索引值，所以这个函数能做 Index()、IndexReflect()、IntSliceIndex()的所有工作，还有上一节的IniIndexSlicer()函数，但没有一行多余的代码，也没有类型开关和类型断言。

SliceIndex()函数并不知道而且也不需要关心切片或者项的类型，实际上，这些对函数来说是透明的，它只知道一个长度信息和它的第

二个参数，也就是个对于任意给定索引值返回一个布尔值的函数，表明这个索引是否是调用者所期望的。

下面是函数调用的4个样例和它们输出的结果。

```
xs := []int{2, 4, 6, 8}
ys := []string{ " C " , " B " , " K " , " A " }
fmt.Println(
    SliceIndex(len(xs), func(i int) bool { return xs[i] == 5 }),
    SliceIndex(len(xs), func(i int) bool { return xs[i] == 6 }),
    SliceIndex(len(ys), func(i int) bool { return xs[i] == " Z " }),
    SliceIndex(len(ys), func(i int) bool { return xs[i] == " A " })))
```

```
-1 2 -1 3
```

传给SliceIndex()的第二个参数的匿名函数是一个闭包，所以它们引用的xs和ys切片必须和这个函数被创建的地方在同一作用域（Go语言标准库里的sort.Search()函数使用同样的技术）。

实际上，SliceIndex()就是一个能直接处理切片的通用函数。

```
i := SliceIndex(math.MaxInt32,
    func(i int) bool { return i > 0 && i%27 == 0 && i%51 == 0 })
fmt.Println(i)
```

```
459
```

上面的代码使用了SliceIndex()来查找能被27和51整除的最小自然数，这种做法有些微妙。SliceIndex()函数从0开始遍历到math.MaxInt32，每一次遍历，它都调用匿名函数，一旦匿名函数返回true，SliceIndex()函数就马上将当前的值返回，这个值就是我们要寻找的自然数。

除查找未排序切片外，另一种有用的场景是过滤掉不关心的数据。下面是一个高阶过滤函数，通过匿名函数来判断传入的`[]int`切片的某项是保留还是丢弃。

```
readings := []int{4, -3, 2, -7, 8, 19, -11, 7, 18, -6}
even := IntFilter(readings, func(i int) bool { return i%2 == 0 })
fmt.Println(even)
```

```
[4 2 8 18 -6]
```

这里，我们过滤掉所有的奇数。

```
func IntFilter(slice []int, predicate func(int) bool) []int {
    filtered := make([]int, 0, len(slice))
    for i := 0; i < len(slice); i++ {
        if predicate(slice[i]) {
            filtered = append(filtered, slice[i])
        }
    }
    return filtered
}
```

`IntFilter()`函数有两个参数，一个是`[]int`切片，另一个是 `predicate()` 函数，返回`true`则表示保留，否则反之。最后返回一个新的切片，包含了过滤后的所有数据。

对切片进行过滤是一个很常用的功能，所以如果`IntFilter()`函数只能处理`[]int`型切片的话那就太可惜了。幸运的是，使用我们设计 `SliceIndex()` 函数的那种技术，我们完全可以创建一个通用的过滤函数。

```
func Filter(limit int, predicate func(int) bool, appender func(int)) {
    for i := 0; i < limit; i++ {
```

```

        if predicate(i) {
            appender(i)
        }
    }
}

```

和SliceIndex()函数一样，Filter()函数也不知道自己所操作的数据实际是什么，Filter()的过滤和追加功能依赖于它传进来的predicate()函数和appender()函数。

```

readings := []int{4, -3, 2, -7, 8, 19, -11, 7, 18, -6}
even := make([]int, 0, len(readings))
Filter(len(readings), func(i int) bool { return readings[i]%2 == 0 },
    func(i int) { even = append(even, readings[i]) })
fmt.Println(even)

```

```
[4 2 8 18 -6]
```

这段代码和之前那段代码的处理过程是完全一样的，只是在这里我们必须在Filter()函数外创建一个新的even切片。我们传给Filter()的第一个匿名函数只有一个索引参数，如果切片里索引处的项是个偶数的话就返回true。第二个匿名函数是将索引处对应的项追加到开始时创建的新的切片里去。当匿名函数被传入时even和readings两个切片要在当前作用域里，这样匿名函数才能够捕获到以便访问它们。

```

parts := []string{ " X15 ", " T14 ", " X23 ", " A41 ", " L19 ", "
X57 ", " A63 " }
var Xparts []string
Filter(len(parts), func(i int) bool { return parts[i][0] == 'X' },
    func(i int) { Xparts = append(Xparts, parts[i]) })
fmt.Println(Xparts)

```

```
[X15 X23 X57]
```

注意这里处理的是字符串而不是整数，可见Filter()函数能支持不同的数据类型。

```
var product int64 = 1
Filter(26, func(i int) bool { return i%2 != 0 },
    func(i int) { product *= int64(i) })
fmt.Println(product)
```

```
7905853580625
```

这是最后一个关于过滤的例子，和SliceIndex()函数差不多，Filter()并非只能用来处理切片，这里我们就用它来计算范围[1, 25]内所有奇数的乘积。

纯记忆函数

所谓纯函数就是对同一组输入总是产生相同的结果，不存在任何副作用。如果一个纯函数执行时开销很大而且频繁地使用相同的参数进行调用，我们可以使用记忆功能来降低处理的开销。记忆技术就是保存计算的结果，当执行下一个相同的计算时，我们能够返回保存的结果而不是重复执行一次计算过程。

使用递归来计算斐波纳契数列的开销非常大，而且重复地计算相同的过程，就像之前图 5-2里看到的。这种情况下最容易的解决方法就是使用一个非递归的算法，但为了展示我们如何使用记忆功能，我们先创建一个使用递归的具有记忆功能的斐波纳契函数。

```
type memoizeFunction func(int,...int) interface{}
var Fibonacci memoizeFunction
func init() {
```

```

Fibonacci = Memoize(func(x int, xs...int) interface{} { {
    if x < 2 {
        return x
    }
    return Fibonacci(x-1).(int) + Fibonacci(x-2).(int)
})
}

```

Memoize()函数（很快就会讲到）可以记忆任何传入至少一个 `int` 参数并返回一个`interface{}`的函数。为了方便，我们为这种函数创建了`memoizeFunction`类型，并声明一个 `Fibonacci` 变量用来保存这个类型的函数。然后，在程序的`init()`函数里，我们创建了一个计算斐波纳契数列的匿名函数，并立即将它传给 **Memoize()**函数。相应地，**Memoize()**函数返回一个`memoizeFunction`类型的函数，然后赋值给`Fibonacci`变量。

在这个特定例子里，我们只需传一个参数给`Fibonacci`函数，所以我们可以忽略所有其他传入的整数（即忽略 `xs`，在这个例子里它应该是一个空的切片）。还有，当我们将递归的结果汇总的时候，我们必须使用非检查类型断言将返回值从`interface{}`类型转换成`int`类型。

现在我们可以像其他函数那样使用 `Fibonacci()`，而且得益于记忆功能，它不会重复执行相同的计算过程。

```
fmt.Println( " Fibonacci(45) = " , Fibonacci(45).(int))
```

```
Fibonacci(45) = 1134903170
```

我们使用非检查类型断言将它的`interface{}`类型的返回值转换成`int`（严格来说，这里不需要做转换，因为`fmt`包实现得非常优雅，它会自动处理这些事情，不过我这样写大家可以看到它实际是怎么用的）。

```
func Memoize(function memoizeFunction) memoizeFunction {
```

```

cache := make(map[string]interface{})
return func(x int, xs...int) interface{} {
    key := fmt.Sprint(x)
    for _, i := range xs {
        key += fmt.Sprintf( " ,%d " , i)
    }
    if value, found := cache[key]; found {
        return value
    }
    value := function(x, xs...)
    cache[key] = value
    return value
}
}

```

我们这里用的Memoize()是最核心的函数，它将memoizeFunction类型的函数（函数签名为func(int,...int) interface{}）作为参数然后返回一个相同签名的函数。

我们使用一个映射结构来保存预先计算的结果，映射的键是字符串，值是一个interface{}。映射被Memoize()返回的匿名函数捕获，也就是闭包。映射的键是将所有的整型参数组合并用逗号分隔的字符串（Go语言的映射要求键必须完全支持==和!=操作，字符串符合这个要求，但是切片不可以，参见4.3节）。键准备好之后我们看看是否在映射里有对应的“键/值”对，如果有我们就不需要重复计算，只需简单返回缓存的结果；否则我们就执行传给Memoize()函数的function 函数，再将结果缓存到映射。这样就不需要再次重复计算这个结果。最后，我们返回计算出来的值。

记忆功能对那些开销大的纯函数（不管它们有没有递归）而言是非常有用的，因为它们浪费了大部分的时间来计算一些参数相同的过程。例如，如果我们需要将大量整数转换成罗马数字而大部分这些整数都是重复的，这种情况就可以用 **Memoize()**函数来避免重复的计算。最好统计那些开销大的计算的花费时间（比如使用**time**包或性能测试工具），以便判断是否记忆功能（或任何其他可能的优化）是值得使用的。

```
var RomanForDecimal memoizeFunction
func init() {
    decimals := []int{1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1}
    romans := []string{ " M ", " CM ", " D ", " CD ", " C ", " XC
", " L ", " XL ", " X ",
        " IX ", " V ", " IV ", " I " }
    RomanForDecimal = Memoize(func(x int, xs...int) interface{} { {
        if x < 0 || x > 3999 {
            panic( " RomanForDecimal() only handles integers [0, 3999] " )
        }
        var buffer bytes.Buffer
        for i, decimal := range decimals {
            remainder := x / decimal
            x %= decimal
            if remainder > 0 {
                buffer.WriteString(strings.Repeat(romans[i], remainder))
            }
        }
        return buffer.String()
    })
}
```



```
}
```

`RomanForDecimal`是一个`memoizeFunction`类型的全局变量（当然，只是在他所在的包里，参见第9章），在`init()`函数里创建。`decimals`和`romans`切片是`init()`函数的本地变量，但是只要`RomanForDecimal()`函数还在使用，它们就会一直存在，因为`RomanForDecimal`是一个捕获了这两个变量的闭包。

Go语言的函数和方法真是难以置信的灵活和强大，提供了好几种方法来根据需求实现泛型。

5.7 例子：缩进排序

这一节要讲解的例子就是如何对字符串进行排序。这个函数之所以特别（以及标准库的`sort.Strings()`函数为何无法满足此需求）是因为这个字符串是按照等级来排序的，也就是它们内部缩进的级别（源码在文件`indent_sort/indent_sort.go`里）。

注意 `SortedIndentStrings()`函数有一个很重要的前提就是，字符串的缩进是通过读到的空格和缩进的个数来决定的，所以我们只需处理单字节的空白，而不必考虑怎么处理多字节的空白（如果我们真的想要处理多字节的空白字符，一种比较容易的方法就是，在将它们传入`SortedIndentedStrings()`函数前将字符串里的空白串替换成一个简单的空格或者缩进符，例如，使用`strings.Map()`函数）。

我们来看一下`main`函数和输出结果。为了方便对比，我们将排序前的结果放在左边，排序后的结果放在右边。

```
func main() {  
    fmt.Println( " |      Original      |      Sorted      | " )  
    fmt.Println( " |-----|-----| " )
```

```

sorted := SortedIndentedStrings(original) // 最初是[]string
for i := range original {                // 在全局变量中设置
    fmt.Printf( " |%-19s|%-19s|\n " , original[i], sorted[i])
}

```

Original	Sorted
-----	-----
Nonmetals	Alkali Metals
Hydrogen	Lithium
Carbon	Potassium
Nitrogen	Sodium
Oxygen	Inner Transitionals
Inner Transitionals	Actinides
Lanthanides	Curium
Europium	Plutonium
Cerium	Uranium
Actinides	Lanthanides
Uranium	Cerium
Plutonium	Europium
Curium	Nonmetals
Alkali Metals	Carbon
Lithium	Hydrogen
Sodium	Nitrogen
Potassium	Oxygen

其中，函数SortedIndentedStrings()和它的辅助函数与类型使用到了递归函数、函数引用以及指向切片的指针等。尽管我们很容易看出程

序做了哪些事情，但是要真正实现这个需求还是要有一些思考的。我们使用了本章介绍过的一些 Go 语言的函数特性，还用到了第 4 章介绍过的一些技巧，这些技巧在第 6 章将会更全面地介绍。

在我们给出的参考答案里，最关键的地方就是自定义的 **Entry** 和 **Entries** 类型。对于在原切片里的每一个字符串，我们为它创建一个 **Entry** 的“键/值”结构，键字段是用来排序的，值字段保存原字符串，而 **children** 字段则是该字符串的孩子 **Entry** 切片（**children** 可能为空，如果不为空，它包含的 **Entry** 自身也还可能包含子 **Entry**，以此类推）。

```
type Entry struct {
    key string
    value string
    children Entries
}
type Entries []Entry
func (entries Entries) Len() int { return len(entries) }
func (entries Entries) Less(i, j int) bool {
    return entries[i].key < entries[j].key
}
func (entries Entries) Swap(i, j int) {
    entries[i], entries[j] = entries[j], entries[i]
}
```

sort.Interface 接口定义了 3 个方法 **Len()**、**Less()** 和 **Swap()**。它们的函数签名和 **Entries** 中的同名方法是一样的。这就意味着我们可以使用标准库里的 **sort.Sort()** 函数来对一个 **Entries** 进行排序。

```
func SortedIndentedStrings(slice []string) []string {
    entries := populateEntries(slice)
    return sortedEntries(entries)
```

```
}
```

导出（公有）的`SortIndentedStrings()`函数就做了这个工作，虽然我们已经对它进行了重构，让它把所有东西都传递给辅助函数。函数`populateEntries()`传入一个`[]string`并返回一个对应的`Entries`（`[]Entry` 类型）。而函数`sortedEntries()`需要传入一个`Entries`，然后返回一个排过序的`[]string`（根据缩进的级别进行排序）。

```
func populateEntries(slice []string) Entries {
    indent, indentSize := computeIndent(slice)
    entries := make(Entries, 0)
    for _, item := range slice {
        i, level := 0, 0
        for strings.HasPrefix(item[i:], indent) {
            i += indentSize
            level++
        }
        key := strings.ToLower(strings.TrimSpace(item))
        addEntry(level, key, item, &entries)
    }
    return entries
}
```

`populateEntries()`函数首先以字符串的形式得到给定切片里的一级缩进（如4个空格的字符串）和它占用的字节数，然后创建一个空的`Entries`，并遍历切片里的每一个字符串，判断该字符串的缩进级别，再创建一个用于排序的键。下一步，函数调用自定义函数`addEntry()`，将当前字符串的级别、键、字符串本身，以及指向`entries`的地址作为参数参数，这样`addEntry()`就能创建一个新的`Entry`并能够正确地将它追加到`entries`里去。最后返回`entries`。

```

func computeIndent(slice []string) (string, int) {
    for _, item := range slice {
        if len(item) > 0 && (item[0] == ' ' || item[0] == '\t') {
            whitespace := rune(item[0])
            for i, char := range item[1:] {
                if char != whitespace {
                    return strings.Repeat(string(whitespace), i), i
                }
            }
        }
    }
    return " ", 0
}

```

`computeIndent()`主要是用来判断缩进使用的是什麼字符，例如空格或者缩进符等，以及一个缩进级别占用多少个这样的字符。

因为第一级的字符串可能没有缩进，所以函数必须迭代所有的字符串。一旦它发现某个字符串的行首是空格或者缩进，函数马上返回表示缩进的字符以及一个缩进所占用的字符数。

```

func addEntry(level int, key, value string, entries *Entries) {
    if level == 0 {
        *entries = append(*entries, Entry{key, value, make(Entries, 0)})
    } else {
        addEntry(level-1, key, value,
            &((*entries)[entries.Len()-1].children))
    }
}

```

`addEntry()`是一个递归函数，它创建一个新的Entry，如果这个Entry的level是0，那就直接增加到entries里去，否则，就将它作为另一个Entry的孩子。

我们必须确定这个函数传入的是一个*Entries而不是传递一个entries引用（切片的默认行为），因为我们是要将数据追加到entries里。追加到一个引用会导致无用的本地副本且原来的数据实际上并没有被修改。

如果level是0，表明这个字符串是顶级项，因此必须将它直接追加到*entries。实际上情况要更复杂一些，因为level是相对传入的*entries而言的，第一次调用addEntry()时，*entries是一个第一级的Entries，但函数进入递归后，*entries就可能是某个Entry的孩子。我们使用内置的append()函数来追加新的Entry，并使用*操作符获得entries指针指向的值，这就保证了任何改变对调用者来说都是可见的。新增的Entry包含给定的key和value，以及一个空的子Entries。这是递归的结束条件。

如果level大于0，则我们必须将它追加到上一级Entry的children字段里去，这里我们只是简单地递归调用addEntry()函数。最后一个参数可能是我们目前为止见到的最复杂的表达式了。

子表达式entries.Len() - 1产生一个int型整数，表示*entries指向的Entries值的最后一个条目的索引位置（注意Entries.Len()传入的是一个Entries值而不是*Entries指针，不过Go语言也可以自动对entries指针进行解引用并调用相应的方法，这一点是很优雅的）。完整的表达式（& (...)除外）访问了Entries最后一个Entry的children字段（这也是一个Entries类型）。所以如果把这个表达式作为一个整体，实际上我们是将Entries里最后一个Entry的children字段的内存地址作为递归调用的参数，因为addEntry()最后一个参数是*Entries类型的。

为了帮助大家弄清楚到底发生了什么，下面的代码和上述代码中else代码块中的那个调用是一样的。

```
theEntries := *entries
lastEntry := &theEntries[theEntries.Len()-1]
addEntry(level-1, key, value, &lastEntry.children)
```

首先，我们创建`theEntries`变量用来保存`*entries`指针指向的值，这里没有什么开销因为不会产生复制，实际上`theEntries`相当于一个指向`Entries`值的别名。然后我们取得最后一项的内存地址（即一个指针）。如果不取地址的话就会取到最后一项的副本，这不是我们期望的。最后递归调用`addEntry()`函数，并将最后一项的`children`字段的地址作为参数传递给它。

```
func sortedEntries(entries Entries) []string {
    var indentedSlice []string
    sort.Sort(entries)
    for _, entry := range entries {
        populateIndentedStrings(entry, &indentedSlice)
    }
    return indentedSlice
}
```

当调用`sortedEntries()`函数的时候，`Entries`显示的结构和原先程序输出的字符串是一样的，每一个缩进的字符串都是上一级缩进的子级，而且还可能有下一级的缩进，依次类推。

创建了`Entries`之后，`SortedIndentStrings()`函数调用上面这个函数去生成一个排好序的字符串切片`[]string`。这个函数首先创建一个空的`[]string`用来保存最后的结果，然后对`entries`进行排序。`Entries`实现了`sort.Interface`接口，因此我们可以直接使用`sort.Sort()`函数根据`Entry`的`key`字段来对`Entries`进行排序（这是`Entries.Less()`的实现方式）。这个排序只是作用于第一级的`Entry`，对其他未排序的孩子`Entry`是没有任何影响的。

为了能够对children字段以及children的children等进行递归排序，函数遍历第一级的每一个项并调用populateIndentedStrings()函数，传入这个Entry类型的项和一个指向[]string切片的指针。

切片可以传递给函数并由函数更新内容（如替换切片里的某些项），但是我们这里需要往切片里新增一些数据。Go语言内置的append()函数有时候返回一个新的切片的引用（比如当原先的切片的容量不足时）。所以这里我们用了另一种处理方式，将一个指向切片的指针（也就是指针的指针）作为参数传进去，并将指针指向的内容设置为append()函数的返回结果，这里可能是一个新的切片，也可能是原先的切片（如果我们不使用指针的话，我们只能得到一个对于调用方不可见的本地切片）。另一种办法就是传入切片的值，然后返回append()之后的切片，但是必须将返回的结果赋值给原来的切片变量（例如slice = function(slice)）。不过这么做的话，很难正确地使用递归函数。

```
func populateIndentedStrings(entry Entry, indentedSlice *[]string) {  
    *indentedSlice = append(*indentedSlice, entry.value)  
    sort.Sort(entry.children)  
    for _, child := range entry.children {  
        populateIndentedStrings(child, indentedSlice)  
    }  
}
```

这个函数将项追加到创建的切片，然后对项的孩子进行排序，并递归调用自身对每一个孩子做同样的处理。这就相当于对每一项的孩子以及孩子的孩子等都做了排序，所以整个字符串切片就是已经排好序的了。

至此我们已经讲完了所有 Go语言的内置数据类型和过程式编程，下一章我们在这基础上对Go语言面向对象编程的特色进行讲解，之后

章节还会学习Go语言对并发编程的支持。

5.8 练习

第5章有4个练习。第一个需要修改我们上面的其中一个例子，部分代码在本章中已经呈现过，另外还需要创建一些新的函数。所有的练习都很短，第一个还非常简单，第二个比较直接明了，但第3个和第4个则非常具有挑战性。

(1) 复制 `archive_file_list` 目录到例如 `my_archive_file_list`，然后修改`archive_file_list/archive_file_list.go` 文件：因为我们要实现一个不同的 `ArchiveFileList()` 函数，所以除了 `ArchiveFileListMap()` 将被改名为 `ArchiveFileList()` 这个函数外，其他的代码都要删除掉。然后增加处理 `.tar.bz2` 文件的功能（使用 `bzip2` 压缩的 `tar` 包）。总共需要删除 `main()` 函数的 11 行代码，删除 4 个函数，导入一个额外的包，为 `FunctionForSuffix` 映射增加一项，并在 `TarFileList()` 函数里增加少量的代码。参考答案在 `archive_file_list_ans/archive_file_list.go` 文件里。

(2) 创建一个非递归版本的 `IsPalindrome()` 函数，这个函数在之前的章节里讲过。`palindrome_ans/palindrome.go` 文件里的参考答案只有 10 行代码长，和递归版本在结构上是完全不一样的，不过它只处理 `ASCII` 编码的字符。另外，非递归的 `UTF-8` 版本有 14 行代码左右，和递归的很相似，不过要有点耐心。

(3) 创建一个 `CommonPrefix()` 函数，接受一个 `[]string` 字符串切片并返回切片里所有字符串的共同前缀（如果不存在，就返回一个空的字符串）。参考答案在 `common_prefix/common_prefix.go` 文件里，大概 22 行代码，使用 `[]rune` 来保存字符串，确保当我们遍历时即使字符串里包含非 `ASCII` 编码的字符我们也可以正确地得到一个完整的字符。参

考答案使用一个 `bytes.Buffer` 来构建结果。尽管程序虽然简短，这并不意味着很容易（下一个练习还有其他的例子）。

（4）创建一个 `CommonPathPrefix()` 函数，传入一个保存了路径的字符串切片 `[]string`，并返回一个所有传入路径字符串的公共前缀（同样可能为空），这个前缀由零到多个完整的路径组件组成。参考答案在 `common_prefix/common_prefix.go` 文件里，包含27行代码，用了一个 `[]string` 来保存所有的路径字符串并使用 `filepath.Separator` 来辨别平台特定的路径分隔符，并返回一个 `[]string` 类型的结果，可以使用 `filepath.Join()` 函数将它们组合成一个完整的路径。虽然程序真的很短，但还是很有挑战性的（下面有一些示例）。

这是上面 `common_prefix` 练习3和练习4程序的输出结果。每两行的第一行是一个字符串切片，第二行则是由 `CommonPrefix()` 函数和 `CommonPathPrefix()` 函数产生的公共前缀，以及这两个公共前缀是否相等的标识。

```
$.common_prefix
[ " /home/user/goeg " " /home/user/goeg/prefix "
" /home/user/goeg/prefix/extra " ]
char×path prefix: " /home/user/goeg " == " /home/user/goeg "
[ " /home/user/goeg " " /home/user/goeg/prefix "
" /home/user/prefix/extra " ]
char×path prefix: " /home/user/ " != " /home/user "
[ " /pecan/π/goeg " " /pecan/π/goeg/prefix " " /pecan/π/prefix/extra
" ]
char×path prefix: " /pecan/π/ " != " /pecan/π "
[ " /pecan/π/circle " " /pecan/π/circle/prefix " " /pecan/
π/circle/prefix/extra " ]
char×path prefix: " /pecan/π/circle " == " /pecan/π/circle "
```

```
[ "/home/user/goeg " " /home/users/goeg " " /home/userspace/goeg
" ]
char×path prefix: " /home/user " != " /home "
[ "/home/user/goeg " " /tmp/user " " /var/log " ]
char×path prefix: " / " == " / "
[ "/home/mark/goeg " " /home/user/goeg " ]
char×path prefix: " /home/ " != " /home "
[ " home/user/goeg " " /tmp/user " " /var/log " ]
char×path prefix: " " == " "
```

[1].[表5-1中没有列出内置的print\(\)和println\(\)函数，因为不推荐使用。它们的存在只是为方便Go编译器的实现者，并且可以从语言层面将其删除，请使用fmt.Print\(\)函数代替它（参见3.5节）。](#)

[2].[本书撰写时，gofmt并不支持将行包裹成支持最宽的行，而有时gofmt会将两行或者多行包裹成一个长行。本书的源代码自动提取自可运行的例子和示例程序中，然后插入本书准备打印的PPT文件中，但这样就很难达到每行75个字符的限制。因此，本书使用了gofmt，但是长的行是手动包裹的。](#)

[3].[也可能有别的更为模糊的转换，这些都在Go语言规范文档中有描述（\[golang.org/doc/go-spec.html\]\(http://golang.org/doc/go-spec.html\)）。](#)

[4].[Python程序员可以把interface{}理解成object实例，而Java程序员则可以将其理解成是Object类的实例，虽然与Java的Object类不一样的是interface{}可以同时用于表示自定义类型和内置类型。对于C和C++程序员而言，interface{}相当于一个知道类型信息的void*。](#)

[5].[1968年，由于Edsger Dijkstra的一封信题为“Go-to Statement considered harmful”（\[www.cs.utexas.edu/users/EWD/ewdO2xx/EWD215.PDF\]\(http://www.cs.utexas.edu/users/EWD/ewdO2xx/EWD215.PDF\)）的信，goto语句开始普遍被人鄙视了。](#)

[6].[这个例子的灵感来自Andrew Gerrand的博客：\[nf.id.au/concurrency-patterns-a-source-of-unique-numbe\]\(http://nf.id.au/concurrency-patterns-a-source-of-unique-numbe\)。（这个网址最后确实没有字母](#)

“r”。)

[7].Go语言中的select语句与POSIX中的select()函数无关，后者是用于监控文件描述符的——这跟syscall包中的Select()函数的功能是一样的。

[8].在C++中，析构函数用于回收资源。在Java和Python中，资源的回收是不确定的，因为它们并不能保证何时甚至是否finalizer()或者 del ()方法会被调用。

[9].在一款轻载的AMD-64四核3GHz机器上，我们发现在同样需要处理50个以上分支的情况下，使用映射的速度会稳定的超过switch语句。

第6章 面向对象编程

本章的目的是讲解在Go语言中如何进行面向对象编程。来自于其他过程式编程背景的程序员可能会发现，本章的所有内容都建立在他们所学以及本书前面章节的基础之上。但是来自于其他基于继承到面向对象编程背景（如C++、Java和Python）的程序员可能需要将许多曾经常用的概念和习惯放在一边，特别是继承相关的，因为Go语言的面向对象编程方式与它们的完全不同。

Go语言的标准库大部分情况下提供的都是函数包，但也适当地提供了包含方法的自定义类型。在前面的章节中，我们创建了一些自定义类型（如`regexp.Regexp`和`os.File`）的值，并也调用了它们的方法。此外，我们甚至创建了一些简单的自定义类型，以及相应的方法。例如，支持打印和排序。因此，我们已经熟悉了Go语言类型的基本使用以及类型方法的调用。

本章第一节用非常简短的篇幅描述了一些Go语言面向对象编程中的关键概念。第二节包含了创建无方法的自定义类型的内容。接下来我们往自定义类型中添加了方法，创建了构造函数，以及验证字段数据，总之，讲解了创建一个独立的自定义类型所需的所有基础内容。第三节讲解了接口，这是Go语言实现类型安全的鸭子类型的基础。第四节讲解了结构体，介绍了许多前面章节中未曾涉及的细节。

本章的最后一节给出了3个关于自定义类型的完整示例，它们覆盖了本章前面各节中的大部分内容以及本书中前面章节中的相当一部分内容。其中，第一个例子是一个简单的只包含单值数据类型的自定义

类型，第二个例子是一小部数据类型的集合，第三个例子是一个通用集合类型。

6.1 几个关键概念

Go语言的面向对象之所以与C++、Java以及（较小程度上的）Python这些语言如此不同，是因为它不支持继承。面向对象编程刚流行的时候，继承是它首先被捧吹的最大优点之一。但是历经几十载的实践之后，事实证明该特性也有些明显的缺点，特别是当用于维护大系统时。与其他大部分同时使用聚合和继承的面向对象语言不同的是，Go语言只支持聚合（也叫做组合）和嵌入。为了弄明白聚合与嵌入的区别，让我们看一小段代码。

```
type ColoredPoint struct{
    color.Color // 匿名字段（嵌入）
    x, y        int// 具名字段（聚合）
}
```

这里，`color.Color`是来自`image/color`包的类型，`x`和`y`则是整型。在Go语言的术语中，`color.Color`、`x`和`y`，都是`ColoredPoint`结构体的字段。`color.Color`字段是匿名的（因为它没有变量名），因此是嵌入字段。`x`和`y`字段是具名的聚合字段。如果我们创建一个 `ColoredPoint` 值（例如，`point := ColoredPoint{}`），其字段可以通过`point.Color`、`point.x`和`point.y`来访问。需注意的是，当访问来自于其他包中的类型的字段时，我们只用到了其名字的最后部分，即`Color`而非`color.Color`（我们会在6.2.1.1节、6.3节及6.4节详细讨论这些内容）。

术语“类”（`class`）、“对象”（`object`）以及“实例”（`instance`）在传统的多层次继承式面向对象编程中已经定义的非常清晰，但在Go语言

中我们完全避开使用它们。相反，我们使用“类型”和“值”，其中自定义类型的值可以包含方法。

由于没有继承，因此也就没有虚函数。Go语言对此的支持则是采用类型安全的鸭子类型（**duck type**）。在 Go语言中，参数可以被声明为一个具体类型（例如，`int`、`string`、或者`*os.File`以及`MyType`），也可以是接口（**interface**），即提供了具有满足该接口的方法的值。对于一个声明为接口的参数，我们可以传入任意值，只要该值包含该接口所声明的方法。例如，如果我们有一个值提供了一个`Write([]byte)(int, error)`方法，我们就可以将该值当做一个`io.Writer`（即作为一个满足`io.Writer`接口的值）提供给任何一个需要`io.Writer`参数的函数，无论该值的实际类型是什么。这点非常灵活而强大，特别是当它与 Go语言所支持的访问嵌入字段的方法相结合时。

继承的一个优点是，有些方法只需在基类中实现一次，即可在子类中方便地使用。Go语言为此提供了两种解决方案。其中一种解决方案是使用嵌入。如果我们嵌入了一个类型，方法只需在所嵌入的类型中实现一次，即可在所有包含该嵌入类型的类型中使用 [1]。另一种解决方案是，为每一种类型提供独立的方法，但是只是简单地将包装（通常都只有一行）了功能性作用的代码放进一个函数中，然后让所有类的方法都调用这个函数。

Go语言面向对象编程中的另一个与众不同点是它的接口、值和方法都相互保持独立。接口用于声明方法签名，结构体用于声明聚合或者嵌入的值，而方法用于声明在自定义类型（通常为结构体）上的操作。在一个自定义类型的方法和任何特殊接口之间没有显式的联系。但是如果该类型的方法满足一个或者多个接口，那么该类型的值可以用于任何接受该接口的值的地方。当然，每一个类型都满足空接口（`interface{}`），因此任何值都可以用于声明了空接口的地方。

一种按Go语言的方式思考的方法是，把is-a关系看成由接口来定义，也就是方法的签名。因此，一个满足 `io.Reader` 接口（即有一个签名为 `Read([]byte)(int, error)`的方法）的值就叫做 `Reader`，这并不是因为它是什么（一个文件、一个缓冲区或者一些其他自定义类型），而是因为它提供了什么方法，在这里是`Read()`方法。如图6-1中的解释。而has-a关系可以使用聚合或者嵌入特定类型值的结构体来表达，这些类型构成自定义类型。

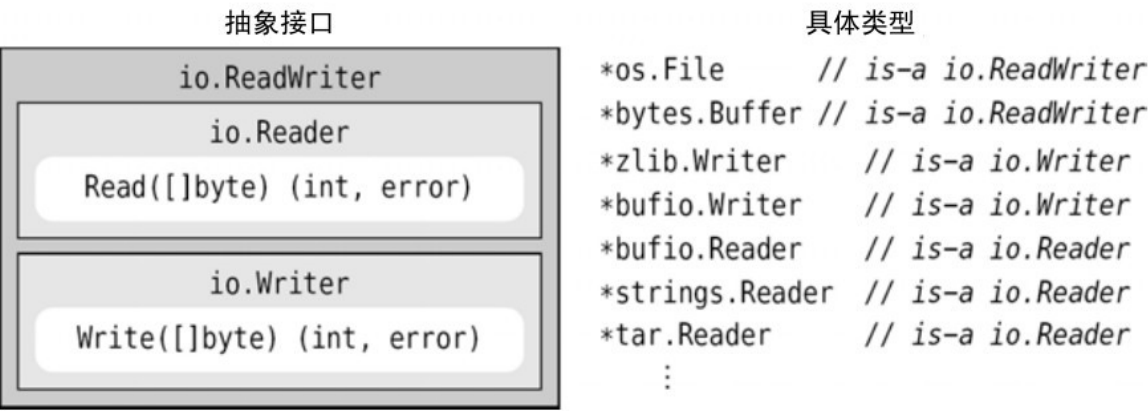


图6-1 用于读写字节切片的接口和类型

虽然没法为内置类型添加方法，但可以很容易地基于内置类型创建自定义的类型，然后为其添加任何我们想要的方法。该类型的值可以调用我们提供的方法，同时也可以与它们底层类型提供的任何函数、方法以及操作符一起使用。例如，假设我们有个类型声明为 `type Integer int`，我们可以不拘形式地使用整型的+操作符将这两种类型的值相加。并且，一旦我们有了一个自定义类型，我们也可以添加自定义的方法。例如， `func (i Integer) Double() Integer{ return i * 2 }`，稍后将会看到（参见6.2.1节）。

基于内置类型的自定义类型不但容易创建，运行时效率也非常高。将基于内置类型的自定义类型与该内置类型相互转换无需耗费运行时代价，因为这种转换能够在编译时高效完成。鉴于此，要使用自

定义类型的方法时将内置类型“升级”成自定义类型，或者要将一个类型传入给一个只接收内置类型参数的函数时将自定义类型“降级”成内置类型，都是非常实用的做法。我们在前文中曾看过一个“升级”的例子，在那里我们将一个[]string类型转换成一个FoldedStrings类型（参见4.2.4节）的值，在本章末尾我们讲解到Count类型的时候我们会举一个“降级”的例子。

6.2 自定义类型

自定义类型使用Go语言的如下语法创建：

```
type typeName typeSpecification
```

typeName可以是一个包或者函数内唯一的任何合法的Go标识符。typeSpecification可以是任何内置的类型（如string、int、切片、映射或者通道）、一个接口（参见6.3节）、一个结构体（参见前面章节，本书后面将介绍更多相关内容，参见6.4节）或者一个函数签名。

在有些情况下创建一个自定义类型就足够了，但有些情况下我们需要给自定义类型添加一些方法来让它更实用。下面是一些没有方法的自定义类型例子。

```
type Count int
```

```
type StringMap map[string]string
```

```
type FloatChan chan float64
```

这些自定义类型就其自身而言，虽然使用这样的类型可以提升程序的可读性，同时也可以在后面改变其底层类型，但是没一个看起来有用，因此只把它们当做基本的抽象机制。

```
var i Count = 7
```

```
i++
```

```
fmt.Println(i)
sm := make(StringMap)
sm[ " key1 " ] = " value1 "
sm[ " key2 " ] = " value2 "
fmt.Println(sm)
fc := make(FloatChan, 1)
fc <- 2.29558714939
fmt.Println(<-fc)
```

```
8
map[key2:value2 key1:value1]
2.29558714939
```

像Count、StringMap和FloatChan这样的类型，它们是直接基于内置类型创建的，因此可以拿来当做内置类型一样使用。例如，我们可以使用内置的append()函数来操作type StringSlice []string类型。但是如果将其传递给一个接受其底层类型的函数，就必须先将其转换成底层类型（无需成本，因为这是在编译时完成的）。有时，我们可能需要进行相反的操作，将一个内置类型的值升级成一个自定义类型的值，以使用其自定义类型的方法。我们已经见过一个这样的例子，在SortFoldedStrings()函数中将一个[]string 转换成一个FoldedStrings值（参见4.2.4节）。

```
type RuneForRuneFunc func(rune) rune
```

当使用高阶函数（参见 5.6.7 节）时，通过自定义类型来声明我们要传入的函数的签名更为方便。这里我们声明了一个接收和返回rune值的函数签名。

```
var removePunctuation RuneForRuneFunc
```

上面创建的removePunctuation变量引用一个RuneForRuneFunc类型的函数（即其签名为func(rune) rune）。与所有Go变量一样，它也被自

动初始化为零值，因此在这里它被初始化成`nil`值。

```
phrases := []string{ " Day; dusk, and night. " , " All day long " }
removePunctuation = func(char rune) rune {
    if unicode.Is(unicode.Terminal_punctuation, char){
        return -1
    }
    return char
}
processPhrases(phrases, removePunctuation)
```

这里我们创建了一个匹配 `RuneForRuneFunc` 签名的匿名函数，并将其传给自定义的`processPhrases()`函数。

```
func processPhrases(phrases []string, function RuneForRuneFunc) {
    for _, phrase := range phrases {
        fmt.Println(strings.Map(function, phrase))
    }
}
```

```
Day dust and night
All day long
```

对读者来说，将 `RuneForRuneFunc` 当成一个类型而非底层的 `func(rune) rune` 更为有意义，同时它也提供了一些抽象。（`strings.Map()` 函数已在第3章中讲解过。）

基于内置类型或者函数签名创建自定义的类型非常有用，但对我们来说还远远不够。我们需要的是自定义的方法，即下一节的内容。

[6.2.1 添加方法](#)

方法是作用在自定义类型的值上的一类特殊函数，通常自定义类型的值会被传递给该函数。该值可以以指针或者值的形式传递，这取决于方法如何定义。定义方法的语法几乎等同于定义函数，除了需要在 `func` 关键字和方法名之间必须写上接收者（写入括号中）之外，该接收者既可以以该方法所属于的类型的形式出现，也可以以一个变量名及类型的形式出现。当调用方法的时候，其接收者变量被自动设为该方法调用所对应的值或者指针。

我们可以为任何自定义类型添加一个或者多个方法。一个方法的接收者总是一个该类型的值，或者只是该类型值的指针。然而，对于任何一个给定的类型，每个方法名必须唯一。唯一名字要求的结果是，我们不能同时定义两个相同名字的方法，让其中一个的接收者为指针类型而另一个为值类型。另一个结果是，不支持重载方法，也就是说，不能定义名字相同但是不同签名的方法。一种提供等价方法的方式是使用可变参数（也就是说，接受可变数目参数，参见本书的第5.6节）。不过，Go语言推荐的方式是使用名字唯一的函数。例如，`strings.Reader` 类型提供 3 个不同的方法：`strings.Reader.Read()`、`strings.Reader.ReadByte()`和`strings.Reader.ReadRune()`。

```
type Count int
func (count *Count) Increment() { *count++ }
func (count *Count) Decrement() { *count-- }
func (count Count) IsZero() bool { return count == 0 }
```

这个简单的基于整型的自定义类型支持3个方法，其中前两个声明为接受一个指针类型的接收者（**receiver**，也就是方法施加的目标对象），因为这两个函数都修改了它们的值 [2]。

```
var count Count
i := int(count)
count.Increment()
```

```
j := int(count)
count.Decrement()
k := int(count)
fmt.Println(count, i, j, k, count.IsZero())
```

```
0 0 1 0 true
```

上面的代码片段展示了 `Count` 类型的实际使用。它看起来没什么，但我们会将其用于本章的第4节。

让我们再稍微多看一个更详细的自定义类型，这回是基于一个结构体定义的（我们会在6.3节中回来再看这个例子）。

```
type Part struct {
    Id    int           // 具名字段（聚合）
    Name string      // 具名字段（聚合）
}
func (part *Part) LowerCase() {
    part.Name = strings.ToLower(part.Name)
}
func (part *Part) UpperCase() {
    part.Name = strings.ToUpper(part.Name)
}
func (part Part) String() string {
    return fmt.Sprintf( "<<%d %q>> ", part.Id, part.Name)
}
func (part Part) HasPrefix(prefix string) bool {
    return strings.HasPrefix(part.Name, prefix)
}
```

为了演示它是如何工作的，我们创建了接收者为值类型的`String()`和`HasPrefix()`方法。当然，传值的话无法修改原始数据，而传递指针的话可以。

```
part := Part{5, " wrench " }
part.UpperCase()
part.Id += 11
fmt.Println(part, part.HasPrefix( " w " ))
```

```
«16 " WRENCH " »false
```

当创建的自定义类型是基于结构体时，我们可以使用其名字及一对大括号包围的初始值来创建该类型的值。（我们在下一节将看到，Go语言提供了一种语法，让我们只提供想要的值，而让Go自己去初始化剩余的值。）

一旦创建了 `part` 值，我们可以在其上调用方法（如 `Part.UpperCase()`），访问它导出的（公开的）字段（如 `Part.Id`），以及安全地打印它，因为如果自定义的类型中定义了 `String()` 方法，Go语言的打印函数足够智能会自动调用该方法进行打印。

类型的方法集是指可以被该类型的值调用的所有方法的集合。

一个指向自定义类型的值的指针，它的方法集由为该类型定义的所有方法组成，无论这些方法接受的是一个值还是一个指针。如果在指针上调用一个接受值的方法，Go语言会聪明地将该指针解引用，并将指针所指的底层值作为方法的接收者。

一个自定义类型值的方法集则由为该类型定义的接收者类型为值类型的方法组成，但是不包括那些接收者类型为指针的方法。但这种限制通常并不像这里所说的那样，因为如果我们只有一个值，仍然可以调用一个接收者为指针类型的方法，这可以借助于 Go语言传值的地址的能力实现，前提是该值是可寻址的（即它是一个变量、一个解引

用指针、一个数组或切片项，或者结构体中的一个可寻址字段）。因此，假设我们这样调用`value.Method()`，其中`Method()`需要一个指针接收者，而 `value` 是一个可寻址的值，Go语言会把这个调用等同于`(&value).Method()`。

`*Count`类型的方法集包含3个方法：`Increment()`、`Decrement()`和`IsZero()`。然而`Count`类型的方法集则只有一个方法：`IsZero()`。所有这些方法都可以在`*Count()`上调用。同时，正如我们在前面的代码片段中所看到的，只要`Count`值是可寻址的，这些函数也可以在`Count`值上调用。`*Part`类型的方法集包含4个方法：`LowerCase()`、`UpperCase()`、`String()`和`HasPrefix()`，而 `Part` 类型的方法集则只包含 `String()`和`HasPrefix()`方法。然而，`LowerCase()`和`UpperCase()`函数也可以作用于可寻址的`Part`值，正如我们在上面代码片段中所看到的。

将方法的接收者定义为值类型对于小数据类型来说是可行的，如数值类型。这些方法不能修改它们所调用的值，因为只能得到接收者的一份副本。如果我们的数据类型的值很大，或者需要修改该值，则需要让方法接受一个指针类型的接收者。这样可以使得方法调用的开销尽可能的小（因为接收者是以32位或者64位的形式传递，无论调用该方法的值多大）。

6.2.1.1 重写方法

本章末尾我们将看到，可以创建包含一个或者多个类型作为嵌入字段的自定义结构体（参见6.4节）。这种方法非常方便的一点是，任何嵌入类型中的方法都可以当做该自定义结构体自身的方法被调用，并且可以将其内置类型作为其接收者。

```
type Item struct {  
    id          string    // 具名字段（聚合）  
    price       float64   // 具名字段（聚合）  
    quantity    int       // 具名字段（聚合）  
}
```

```

}
func (item *Item) Cost() float64 {
    return item.price * float64(item.quantity)
}
type SpecialItem struct {
    Item          // 匿名字段（嵌入）
    catalogId     int // 具名字段（聚合）
}

```

这里，`SpecialItem`嵌入了一个`Item`类型。这意味着我们可以在一个`SpecialItem`上调用`Item`的`Cost()`方法。

```

special := SpecialItem{Item{ " Green ", 3, 5}, 207}
fmt.Println(special.id, special.price, special.quantity, special.catalogId)
fmt.Println(special.Cost())

```

```

Green 3 5 207
15

```

当调用 `special.Cost()` 的时候，`SpecialItem` 类型没有它自身的`Cost()`方法，Go语言使用 `Item.Cost()`方法。同时，传入其嵌入的`Item` 值，而非整个调用该方法的`SpecialItem`值。

稍后我们将看到，如果嵌入的`Item`中有任何字段与`SpecialItem`的字段同名，那么我们仍然可以通过使用类型作为该名字的一部分来调用`Item`的字段。例如，`special.Item.price`。

同时也可以自定义的结构体中创建与所嵌入的字段中的方法同名的方法，来覆盖被嵌入字段中的方法。例如，假设我们有一个新的`item`类型：

```

type LuxuryItem struct {
    Item // 匿名字段（嵌入）
    markup float64 // 具名字段（聚合）
}

```



```
}
```

如上所述，如果我们在`LuxuryItem`上调用`Cost()`方法，就会使用嵌入的`Item.Cost()`方法，就像`SpecialItems`中一样。下面提供了3种不同的覆盖嵌入方法的实现（当然，只使用了其中的一种！）。

```
/*  
func (item *LuxuryItem) Cost() float64 { // 没必要这么冗长！  
    return item.Item.price * float64(item.Item.quantity) * item.markup  
}  
func (item *LuxuryItem) Cost() float64 { // 没必要的重复！  
    return item.price * float64(item.quantity) * item.markup  
}  
*/  
func (item *LuxuryItem) Cost() float64{ // 完美  
    return item.Item.Cost() * item.markup  
}
```

最后一个实现充分利用了嵌入的`Cost()`方法。当然，如果我们不希望这样做，也没必要使用嵌入类型的方法来重写方法（嵌入字段将在稍后讲解结构体时讲到，参见6.4节）。

6.2.1.2 方法表达式

就像我们可以对函数进行赋值和传递一样，我们也可以对方法表达式进行赋值和传递。方法表达式是一个必须将方法类型作为第一个参数的函数。（在其他语言中常常使用术语“未绑定方法”（`unbound method`）来表示类似的概念。）

```
asStringV := Part.String           // 有效签名: func(Part) string  
sv := asStringV(part)  
hasPrefix := Part.HasPrefix       // 有效签名: func(Part, string) bool  
asStringP := (*Part).String       // 有效签名: func(*Part) string
```

```
sp := asStringP(&part)
lower := (*Part).LowerCase    // 有效签名: func(*Part)
lower(&part)
fmt.Println(sv, sp, hasPrefix(part, " w "), part)
```

```
«16 " WRENCH " » «16 " WRENCH " » true «16 " wrench " »
```

这里我们创建了 4 个方法表达式：`asStringV()`接受一个 `Part` 值作为其唯一的参数，`hasPrefix()`接受一个 `Part` 值作为其第一个参数以及一个字符串作为其第二个参数，`asStringP()`和`lower()`都接受一个`*Part`值作为其唯一参数。

方法表达式是一种高级特性，在关键时刻非常有用。

目前为止我们所创建的自定义类型都有一个潜在的致命错误。没有一个自定义类型可以保证它们初始化的数据是有效的（或者说强制有效），也没有任何方法可以保证这些类型的数据（或者说结构体类型中的字段）不会被赋值为非法数据。例如，`Part.Id`和`Part.Name`字段可以设置为任何我们想设置的值。但如果我们想为其设置限制呢？例如，只允许ID为正整数，而且只允许名字为某固定格式？我们将在下一节讨论该问题，届时我们会创建一个小而全的其字段经验证的自定义类型。

6.2.2 验证类型

对于许多简单的自定义类型来说，没必要进行验证。例如，我们可能这样定义一个类型`type Point {x, y int}`，其中任何`x`和`y`值都是合法的。此外，由于Go语言保证初始化所有变量（包括结构体的字段）为它们的零值，因此显式的构造函数就是多余的。

对于其零值构造函数不能满足条件的情况下，我们可以创建一个构造函数。Go语言不支持构造函数，因此我们必须显式地调用构造函数。为了支持这些，我们必须假设该类型有一个非法的零值，同时提供一个或者多个构造函数用于创建合法的值。

当碰到其字段必须被验证时，我们也可以使用类似的方法。我们可以将这些字段设为非导出的，同时使用导出的访问函数来做一些必要的验证。[\[3\]](#)

让我们来看一个短小但完整的自定义类型来解释这些要点。

```
type Place struct {
    latitude, longitude float64
    Name                string
}

func New(latitude float64, longitude float64, name string) *Place {
    return &Place{ saneAngle(0, latitude), saneAngle(0, longitude),
name }
}

func (place *Place) Latitude() float64 { return place.latitude }
func (place *Place) SetLatitude(latitude float64) {
    place.latitude = saneAngle(place.latitude, latitude)
}

func (place *Place) Longitude() float64 { return place.longitude }
func (place *Place) SetLongitude(longitude float64) {
    place.longitude = saneAngle(place.longitude, longitude)
}

func (place *Place) String() string {
    return fmt.Sprintf( " (%.3f°, %.3f°) %q " , place.latitude,
place.longitude, place.Name)
```

```

}
func (original *Place) Copy() *Place {
    return &Place{ original.latitude, original.longitude, original.Name }
}

```

类型Place是导出的（从place包中），但是它的latitude和longitude字段是非导出的，因为它们需要验证。我们创建了一个构造函数New()来保证总是能够创建一个合法的*place.Place。Go语言的惯例是调用New()构造函数，如果定义了多个构造函数，则调用以“New”开头的那些。（由于有点跑题，我们还没给出saneAngle()函数。它接受一个旧的角度值和一个新的角度值，如果新值在其范围内则返回新值。否则返回旧值。）同时通过提供未导出字段的getter和setter函数，我们可以保证只为其设置合法的值。

String()方法的定义意味着*Place值满足fmt.Stringer接口，因此*Place会按照我们想要的方式而非Go语言的默认格式进行打印。同时我们也提供了一个Copy()方法，但并未为它提供任何验证机制，因为我们知道被复制的原始值是合法的。

```

newYork := place.New(40.716667, -74, " New York ") // newYork是一个*Place

```

```

fmt.Println(newYork)
baltimore := newYork.Copy() // baltimore是一个*Place
baltimore.SetLatitude(newYork.Latitude() - 1.43333)
baltimore.SetLongitude(newYork.Longitude() - 2.61667)
baltimore.Name = " Baltimore "
fmt.Println(baltimore)
(40.717°, -74.000°) " New York "
(39.283°, -76.617°) " Baltimore "

```

我们将Place类型放在place包中，并调用place.New()函数来创建一个*Place的值。一旦创建了一个*Place，我们就可以像调用任何标准库中自定义类型的方法一样调用该*Place值的方法。

6.3 接口

在 Go语言中，接口是一个自定义类型，它声明了一个或者多个方法签名。接口是完全抽象的，因此不能将其实例化。然而，可以创建一个其类型为接口的变量，它可以被赋值为任何满足该接口类型的实际类型的值。

interface{}类型是声明了空方法集的接口类型。无论包含不包含方法，任何一个值都满足 interface{}类型。毕竟，如果一个值有方法，那么其方法集包含空的方法集以及它实际包含的方法。这也是 interface{}类型可以用于任意值的原因。我们不能直接在一个以interface{}类型值传入的参数上调用方法（虽然该值可能有一些方法），因为该值满足的接口没有方法。因此，通常而言，最好以实际类型的形式传入值，或者传入一个包含我们想要的方法的接口。当然，如果我们不为有方法的值使用接口类型，我们就可以使用类型断言（参见5.1.2节）、类型开关（参见5.2.2.2节）或者甚至是反射（参见9.4.9节）等方式来访问方法。

这里有个非常简单的接口。

```
type Exchanger interface {  
    Exchange()  
}
```

Exchanger接口声明了一个方法Exchange()，它不接受输入值也不返回输出。根据Go语言的惯例，定义接口时接口名字需以er结尾。定义

只包含一个方法的接口是非常普遍的。例如，标准库中的`io.Reader`和`io.Writer`接口，每一个都只声明了一个方法。需注意的是，接口实际上声明的是一个API（Application Programming Interface，程序编程接口），即0个或者多个方法，虽然并不明确规定这些方法所需的功能。

一个非空接口自身并没什么用处。为了让它发挥作用，我们必须创建一些自定义的类型，其中定义了一些接口所需的方法 [4]。这里有两个自定义类型。

```
type StringPair struct { first, second string }
func (pair *StringPair) Exchange() {
    pair.first, pair.second = pair.second, pair.first
}
type Point [2]int
func (point *Point) Exchange() { point[0], point[1] = point[1], point[0]
}
```

自定义的类型`StringPair`和`Point`完全不同，但是由于它们都提供了`Exchange()`方法，因此两个都能够满足`Exchanger`接口。这意味着我们可以创建`StringPair`和`Point`值，并将它们传给接受`Exchanger`的函数。

需注意的是，虽然`StringPair`和`Point`类型都能够满足`Exchanger`接口，但是我们并没有这样显式地声明，我们也没有写任何`implements`或者`inherits`语句。`StringPair`和`Point`类型提供了该接口所声明的方法（在这里只有一个方法），这一事实足够让Go语言知道它们满足该接口。

方法的接收者声明为指向其类型的指针，以便我们可以修改调用该方法的（指针所指向的）值。

虽然Go语言足够聪明会以合理的方式打印自定义类型，我们更希望通过它们的字符串表示来控制打印。这可以很容易地通过为其添加一个满足`fmt.Stringer`接口的方法来实现，即一个满足签名`String()string`的方法。

```
func (pair StringPair) String() string {
    return fmt.Sprintf( " %q+%q ", pair.first, pair.second)
}
```

该方法返回一个字符串，该字符串由两个用双引号包围的字符串组合而成，中间用“+”号连接。该方法定义好后，Go语言的fmt包的打印函数就会使用它来打印StringPair值。当然也包括*StringPair的值，因为Go语言会自动将其解引用，以得到其所指向的值。

下面有个代码片段，展示了一些Exchanger值的创建、它们对Exchange()方法的调用，以及对接受Exchanger值的自定义方法exchangeThese()函数的调用。

```
jekyll := StringPair{ " Henry ", " Jekyll " }
hyde := StringPair{ " Edward ", " Hyde " }
point := Point{5, -3}
fmt.Println( " Before: ", jekyll, hyde, point)
jekyll.Exchange()    // 当做: (&jekyll).Exchange()
hyde.Exchange()      // 当做: (&hyde).Exchange()
point.Exchange()     // 当做: (&point).Exchange()
fmt.Println( " After #1: ", jekyll, hyde, point)
exchangeThese(&jekyll, &hyde, &point)
fmt.Println( " After #2: ", jekyll, hyde, point)
```

```
Before: " Henry " + " Jekyll " " Edward " + " Hyde " [5 -3]
After #1: " Jekyll " + " Henry " " Hyde " + " Edward " [-3 5]
After #2: " Henry " + " Jekyll " " Edward " + " Hyde " [5 -3]
```

上面所创建的变量都是值，然而Exchange()方法需要的是一个指针类型接收者。我们之前也注意到，这并不是什么问题，因为当我们调用一个需要指针参数的方法而实际传入的只是可寻址的值时，Go语言会智能地将该值的地址传给方法。因此，在上面的代码片段中，

`jeekyll.Exchange()`会自动被当做`(&jeekyll).Exchange()`用，其他的方法调用情况也类似。

在调用`exchangeThese()`函数的时候，我们必须显式地传入值的地址。假如我们传入的是`StringPair`类型的值`hyde`，Go编译器会发现`StringPair`不能满足`Exchanger`接口，因为在`StringPair`接收者上并未定义方法，从而停止编译并报告错误。然而，如果我们传入一个`*StringPair`（如`&hyde`），编译就能成功。之所以这样，是因为有一个接受`*StringPair`接收者的方法`Exchange()`，也意味着`*StringPair`满足`Exchanger`接口。

这里是`exchangeThese()`函数。

```
func exchangeThese(exchangers...Exchanger) {
    for _, exchanger := range exchangers {
        exchanger.Exchange()
    }
}
```

这个函数并不关心我们传入的是什么类型（实际上我们传入的是两个`*StringPair` 值和一个`*Point` 值），只要它满足 `Exchanger` 接口即可（编译器检查），所以这里用的鸭子类型是类型安全的。

正如我们在定义`StringPair.String()`方法以满足`fmt.Stringer`接口时所看到的一样，除了满足我们自定义的接口之外，我们也可以满足标准库中或者任何其他我们需要的接口。另一个例子`io.Reader`接口，它声明了`Read([]byte)(int, error)`方法签名，当被调用时，它会将调用它的值的数据写入给定的`[]byte` 切片中。这种写是破坏式的，也就是说，写入的每一字节都从其调用处被删除。

```
func (pair *StringPair) Read(data []byte) (n int, err error) {
    if pair.first == " " && pair.second == " " {
        return 0, io.EOF
    }
}
```



```

    }
    if pair.first != " " {
        n = copy(data, pair.first)
        pair.first = pair.first[n:]
    }
    if n < len(data) && pair.second != " " {
        m := copy(data[n:], pair.second)
        pair.second = pair.second[m:]
        n += m
    }
    return n, nil
}

```

只要实现了这个Read()方法，StringPair类型就满足了io.Reader接口的定义。因此，现在StringPair（或者准确地说是*StringPair，因为有些方法需要指针类型的接收者）既是Exchanger和fmt.Stringer，也是io.Reader。不用说，*StringPair肯定实现了这些接口所定义的所有方法了。当然，我们也可以添加更多的方法以满足更多我们想要的接口。

该方法使用了内置的copy()函数（参见4.2.3节）。该函数可以用于将数据从一个切片复制到另一个切片。但是这里我们以另外一种形式使用它，将字符串拷进[]byte。函数copy()复制的数据不会超出目标[]byte的容量，同时返回其复制的字节数。自定义的StringPair.Read()方法从其第一个字符串写数据（同时将已写的数据删除），然后对第二个字符串做同样的操作。如果两个字符串都是空的，则方法返回一个字节数0以及io.EOF。值得一提的是，如果第二条if语句的声明无条件地执行了，而第三个if语句的第二个条件删除了，该方法仍能够完美地运行，只是损失了一些（也许是微不足道的）效率。

这里有必要使用一个指针接收者，因为 `Read()` 方法会修改调用它的值。通常而言，除小数据外，我们更倾向于使用指针接收者，因为传指针比传值更为高效。

定义了 `Read()` 方法之后，我们就可以使用它了。

```
const size = 16
robert := &StringPair{ " Robert L. ", " Stevenson " }
david := StringPair{ " David ", " Balfour " }
for _, reader := range []io.Reader{robert, &david} {
    raw, err := ToBytes(reader, size)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf( " %q\n ", raw)
}
```

```
" Robert L.Stevens "
" DavidBalfour "
```

该代码片段创建了两个 `io.Reader`。由于我们实现 `StringPair.Read()` 方法的时候接收者是一个指针类型，因此只有 `*StringPair` 类型才能满足 `io.Reader()` 接口，而 `StringPair` 值不能满足。对于第一个 `StringPair`，我们创建了它的值，并将 `robert` 变量赋值为指向它的指针，对于第二个 `StringPair`，我们将 `david` 变量赋值为一个 `StringPair` 值，因此在 `[]io.Reader` 切片中使用了它的地址。

一旦变量设置好后，我们就可以迭代它们，对于每一个变量，我们使用自定义的 `ToBytes()` 函数将其数据复制到 `[]byte` 中，然后将其原始字节以双引号括起来的字符串的形式打印出来。

该 `ToBytes()` 函数接受一个 `io.Reader` (即任何包含签名为 `Read([]byte)(int,error)` 的方法的值，例如 `*os.File` 值) 和一个大小限制，同

时返回一个包含所读数据的[]byte切片和一个error值。

```
func ToBytes(reader io.Reader, size int) ([]byte, error) {  
    data := make([]byte, size)  
    n, err := reader.Read(data)  
    if err != nil {  
        return data, err  
    }  
    return data[:n], nil // 清除无用的字节  
}
```

就像我们之前所看到的exchangeThese()函数一样，该函数不知道也不关心所传入值的具体类型，只要它是某种类型的io.Reader。

如果数据读成功，该数据切片会被重新切片以将其长度减至实际所读数据的字节数。如果我们不这样做，并且其预设的大小值太大，那么最终得到的数据也会包含所读数据之外的字节（每个字节的值为0x00）。例如，如果不重新切片，david变量的值可能是这样的 "DavidBalfour\x00\x00\x00\x00"。

需注意的是，接口和满足该接口的任何类型之间没有显式的连接。我们无需声明一个自定义的类型inherits、extends或者implements一个接口，只需给某个类型定义所需的方法就足够了。这使得Go语言非常灵活。我们可以很容易地随时添加新接口、类型以及方法，而无需破坏继承树。

接口嵌入

Go语言的接口（也包括我们将在下一节看到的结构体）对嵌入的支持非常好。接口可以嵌入其他接口，其效果与在接口中直接添加被嵌入接口的方法一样。让我们以一个简单的例子来解释。

```
type LowerCaser interface {  
    LowerCase()  
}
```

```

}
type UpperCaser interface {
    UpperCase()
}
type LowerUpperCaser interface {
    LowerCaser // 就像在这里写了LowerCase()函数一样
    UpperCaser // 就像在这里写了UpperCase()函数一样
}

```

LowerCaser 接口声明了一个方法 LowerCase(), 它不接受参数, 也没有返回值。UpperCaser 接口也类似。而 LowerUpperCaser 接口则将这两个接口嵌套进来。这也意味着对于一个具体的类型, 如果要满足 LowerUpperCaser接口, 就必须定义LowerCase()和UpperCase()方法。

这个小例子的嵌入可能看起来没多大优势。然而, 如果我们要为前两个接口添加额外的方法 (例如, LowerCaseSpecial() 方法和 UpperCaseSpecial()方法), 那么LowerUpperCaser接口也会自动地将其包含进来, 而无需修改自己的代码。

```

type FixCaser interface {
    FixCase()
}
type ChangeCaser interface {
    LowerUpperCaser // 就像在这里写了 LowerCase() 函数和
    UpperCase()函数一样
    FixCaser //就像在这里写了FixCase()函数一样
}

```

这里我们再添加两个接口, 因此现在得到了一个分等级的嵌套接口, 如图6-2所示。

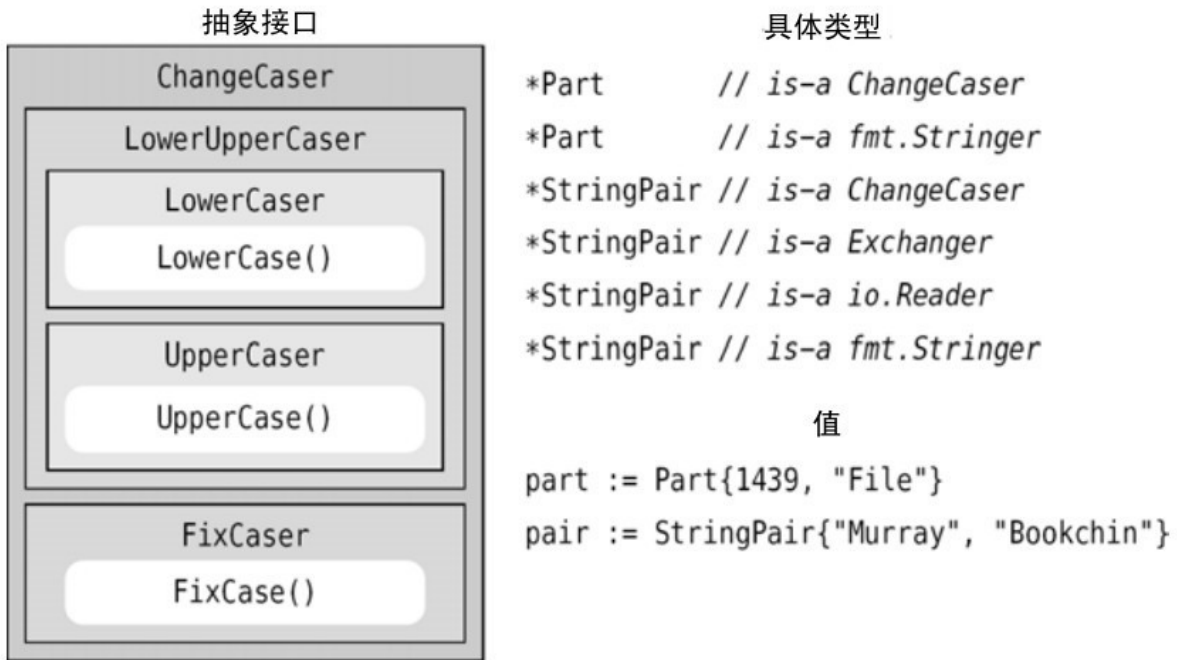


图6-2 Caser接口、类型和示例值

当然，这些接口本身并没多大用处。为了让它们发挥作用，我们需要定义具体的类型来实现它们。

```
func (part *Part) FixCase() {
    part.Name = fixCase(part.Name)
}
```

我们在前面已给出了自定义类型 **Part**（参见 6.2.1 节）。这里，为其添加了一个额外的方法 **FixCase()**，它工作于 **Part** 的 **Name** 字段，就像前文的 **LowerCase()** 和 **UpperCase()** 方法一样。所有这些大小写转换方法都接受一个指针类型的接收者，因为它们需要修改调用它的值。**LowerCase()** 方法和 **UpperCase()** 方法通过标准库来实现，而 **FixCase()** 方法则依赖于自定义的 **fixCase()** 函数。这种简短方法依赖于函数来实现具体功能的模式在 Go 语言中非常普遍。

Part.String() 方法满足标准库中的 **fmt.Stringer** 接口，这意味着任何 **Part**（或者 ***Part**）类型的值都可以使用该方法返回的字符串进行打印。

```

func fixCase(s string) string {
    var chars []rune
    upper := true
    for _, char := range s {
        if upper {
            char = unicode.ToUpper(char)
        } else {
            char = unicode.ToLower(char)
        }
        chars = append(chars, char)
        upper = unicode.IsSpace(char) || unicode.Is(unicode.Hyphen, char)
    }
    return string(chars)
}

```

这个简单的函数返回给定字符串的一份副本，其中除了字符串的首字母及空格或者连字符后面的第一个字母大写之外，其他所有字母都是小写的。例如，给定字符串“lobelia sackville-baggins”，该函数会将其转换成“Lobelia Sackville-Baggins”。

自然，我们可以让所有自定义类型都满足这些大小写转换接口。

```

func (pair *StringPair) UpperCase() {
    pair.first = strings.ToUpper(pair.first)
    pair.second = strings.ToUpper(pair.second)
}

func (pair *StringPair) FixCase() {
    pair.first = fixCase(pair.first)
    pair.second = fixCase(pair.second)
}

```

这里我们为之前所创建的StringPair类型添加了两个方法，使它满足 LowerCaser 、 UpperCaser 和 FixCaser 接口。我们没有列出 StringPair.LowerCase()方法，因为它与StringPair.UpperCase()方法的代码结构完全相同。

*Part 和 *StringPair 两种类型都能够满足 caser 接口，包括 ChangeCaser接口，因为这些类型满足其所有嵌入的接口。它们也同时满足标准库中的 fmt.Stringer 接口。而 *StringPair 类型满足我们的 Exchanger接口以及标准库中的io.Reader接口。

我们并不是强制要求满足每个接口。例如，如果我们选择不实现 StringPair.FixCase() 接口， *StringPair 类型就只能满足 LowerCaser 、 UpperCaser 、 LowerUpperCaser 、 Exchanger 、 fmt.Stringer和io.Reader接口。

下面让我们创建一些值，看看它们的方法。

```
toastRack := Part{8427, " TOAST RACK " }
toastRack.LowerCase()
lobelia := StringPair{ " LOBELIA " , " SACKVILLE-BAGGINS " }
lobelia.FixCase()
fmt.Println(toastRack, lobelia)
```

```
«8427 " toast rack " » " Lobelia " + " Sackville-Baggins "
```

这些方法被调用时其行为如我们所料。但如果我们有一堆这样的值而想在它们之上调用方法呢？下面的做法不太好。

```
for _, x := range []interface{}{&toastRack, &lobelia} { // 不安全！
    x.(LowerUpperCaser).UpperCase() // 未经检查的类型断言
}
```

由于所有的大小写转换方法都会修改调用它的值，因此我们必须使用指向值的指针，因此需要传入指针接收者。

这里所使用的方法有两点缺陷。相对较小的一个缺陷是该未经检查的类型断言是作用于LowerUpperCaser接口的，它比我们实际所需要的接口更泛化。更糟糕的一种做法是使用更为泛化的ChangeCaser接口。但是我们不能使用FixCaser接口，因为它只提供了FixCase()方法。我们应该采用刚好能满足条件的特定接口，这个例子中是UpperCaser接口。该方法最主要的缺陷是使用了一个未经检查的类型断言，可能导致抛出异常！

```
for _, x := range []interface{}{&toastRack, &lobelia} {  
    if x, ok := x.(LowerCaser); ok { // 影子变量  
        x.LowerCase()  
    }  
}
```

上面的代码片段使用了一种更为安全的方式且使用了最合适的特定接口来完成工作，但这相当笨拙。这里的问题是，我们使用的是一个通用的interface{}值的切片，而非一个具体类型的值或者满足某个特殊类型接口的切片。当然，如果所给的都是[]interface{}，那么这种做法是我们所能做到的最好的。

```
for _, x := range []FixCaser { &toastRack, &lobelia } { // 完美的做法  
    x.FixCase()  
}
```

上面代码所示的方式是最好的。我们将切片声明为符合我们需求的FixCaser而不是对原始的interface{}接口做类型检查，从而把类型检查工作交给编译器。

接口的灵活性的另一方面是，它们可以在事后创建。例如，假设我们创建了一些自定义的类型，其中有一些有一个 IsValid() bool 方法。如果后面我们有一个函数需要检查其所接收到的某个值是不是我们定义的，通过检查它是否支持 IsValid()方法来调用该方法，这就很容易做到。


```
type IsValider interface {  
    IsValid() bool  
}
```

首先，我们创建了一个接口，它声明了一个我们希望检查的方法。

```
if thing, ok := x.(IsValider); ok {  
    if !thing.IsValid(){  
        reportInvalid(thing)  
    } else {  
        //...处理有效的thing...  
    }  
}
```

创建了该接口之后，我们现在就可以检查任意自定义类型看它是否提供`IsValid() bool`方法了，如果提供了，我们就调用该方法。

接口提供了一种高度抽象的机制。当某些函数或者方法只关心该传入的值能完成什么功能，而不关心该值的实际类型时，接口允许我们声明一个方法集合，并让这些函数或者方法使用接口参数。本章的后面节中我们将进一步讨论它们的使用（参见6.5.2节）。

6.4 结构体

在Go语言中创建自定义结构体最简单的方式是基于Go语言的内置类型创建。例如，`type Integer int`创建了一个自定义的`Integer`类型，其中我们可以添加自己的方法。自定义类型也可以基于结构体创建，用于聚合和嵌入。这种方式非常有用，因为当值（在结构体中叫做字段）来自不同类型时，它不能存储在一个切片中（除非我们使用

[interface{}))。与C++的结构体相比，Go语言的结构体更接近于C的结构体（例如，它们不是类），并且由于对嵌入的完美支持，它更容易使用。

在前面的章节以及本章中，我们已经看过了很多关于结构体的例子，本书接下来还有更多关于结构体的例子。但是，有些结构体的特性我们还没看到过，因此让我们从一些说明性的例子开始讲解。

```
points := [][]int{{4, 6}, {}, {-7, 11}, {15, 17}, {14, -8}}
for _, point := range points {
    fmt.Printf(" (%d, %d) ", point[0], point[1])
}
```

上面代码片段中的points变量是一个[2]int类型的切片，因此我们必须使用[]索引操作符来获得每一个坐标。（顺便提一下，得益于Go语言的自动零值初始化功能，{}项与{0, 0}项等价。）对于小而简单的数据而言，这段代码能够工作得很好，但还有一种使用匿名结构体的更好的方法。

```
points := []struct{x, y int} {{4, 6}, {}, {-7, 11}, {15, 17},
{14, -8}}
for _, point := range points {
    fmt.Printf(" (%d, %d) ", point.x, point.y)
}
```

在这里，上面的代码片段中的points变量是一个struct{x, y int}结构体。虽然该结构体本身是匿名的，我们仍然可以通过具名字段来访问其数据，这比前面所使用的数组索引更为简便和安全。

结构体的聚合与嵌入

我们可以像嵌入接口或者其他类型的方式那样来嵌入结构体，也就是通过将一个结构体的名字以匿名字段的方式放入另一个结构体中

来实现。（当然，如果我们给内部结构体一个名字，那该结构体就成了一个聚合的具名字段，而非一个嵌入的匿名字段。）

通常一个嵌入字段的字段可以通过使用.（点）操作符来访问，而无需提及其类型名，但是如果外部结构体有一个字段的名字与嵌入的结构体中某个字段名字相同，那么为了避免歧义，我们使用时必须带上嵌入结构体的类型名。

结构体中的每一个字段的名字都必须是唯一的。对于嵌入的（即匿名的）字段，唯一性要求足以保证避免歧义。例如，如果我们有一个类型为 `Integer` 的匿名字段，那么我们还可以包含名字为比如 `Integer2` 或者 `BigInteger` 的字段，因为它们有明显的区别，但却不能包含像 `Matrix.Integer` 或者 `*Integer` 这样的字段，因为这些名字的最后部分字段嵌入的 `Integer` 字段完全一样，而字段的名字的唯一性要求是基于它们的最后部分的。

嵌入值

让我们看一个简单的例子，它涉及了两个结构体。

```
type Person struct {
    Title      string           // 具名字段（聚合）
    Forenames  []string          // 具名字段（聚合）
    Surname    string           // 具名字段（聚合）
}

type Author1 struct {
    Names      Person           // 具名字段（聚合）
    Title      []string          // 具名字段（聚合）
    YearBorn   int              // 具名字段（聚合）
}
```

在前面的章节中，我们看到过许多类似的例子。这里，`Author1` 结构体的字段都是具名的。下面演示了如何使用这些结构体，并给出了

它们的输出（使用一个自定义的`Author1.String()`方法，这里未给出）。

```
author1 := Author1{ Person{ " Mr " , []string{ " Robert " , " Louis " ,  
" Balfour " }, " Stevenson " },  
    []string{ " Kidnapped " , " Treasure Island " }, 1850}  
fmt.Println(author1)  
author1.Names.Title = " "  
author1.Names.Forenames = []string{ " Oscar " , " Fingal " , "  
O'Flahertie " , " Wills " }  
author1.Names.Surname = " Wilde "  
author1.Title = []string{ " The Picture of Dorian Gray " }  
author1.YearBorn += 4  
fmt.Println(author1)
```

```
Stevenson, Robert Louis Balfour, Mr (1850) " Kidnapped " "  
Treasure Island "  
Wilde, Oscar Fingal O'Flahertie Wills (1854) " The Picture of Dorian  
Gray "
```

上面代码开始时创建了一个 `Author1` 值，并将其所有字段都填充上，然后打印。然后，我们更改了该值的字段并再次将其输出。

```
type Author2 struct {  
    Person           // 匿名字段（嵌入）  
    Title    []string // 具名字段（聚合）  
    YearBorn    int     // 具名字段（聚合）  
}
```

为了嵌入一个匿名字段，我们使用了要嵌入类型（或者接口，稍后看到）的名字而未声明一个变量名。我们可以直接访问这些字段的字段（即无需声明类型或者接口名），或者为了与外围结构体的字段的名字区分开，使用类型或者接口的名字访问嵌入字段的字段。

下面给出的Author2结构体嵌入了一个Person结构体作为其匿名字段。这意味着我们可以直接访问Person字段（除非我们需要避免歧义）。

```
author2 := Author2{Person{ " Mr " , []string{ " Robert " , " Louis " ,  
" Balfour " },  
" Stevenson " }, []string{ " Kidnapped " , " Treasure Island " },  
1850}  
fmt.Println(author2)  
author2.Title = []string{ " The Picture of Dorian Gray " }  
author2.Person.Title = " " // 必须使用类型名以消除歧义  
author2.Forenames = []string{ " Oscar " , " Fingal " , " O'Flahertie " ,  
" Wills " }  
author2.Surname = " Wilde " // 等同于 :  
author2.Person.Surname = " Wilde "  
author2.YearBorn += 4  
fmt.Println(author2)
```

上面演示Author1结构体使用的代码在这里重复了一遍，用于演示Author2结构体的使用。它的输出与上例相同（假设我们创建了一个功能与 Author1.String()方法相同的Author2.String()方法）。

通过嵌入Person作为匿名字段，我们所得到的效果与直接添加Person结构体的字段所得到的效果几乎相同。但也不全是，因为如果我们把这些字段添加进来，就得到两个Title字段了，从而不能通过编译。

创建 Author2 值的效果等价于创建 Author1的效果，除非需要消除歧义（author2.Persion.Title与author2.Title的歧义），我们可以直接引用Person中的字段（例如， author2.Forenames）。

嵌入带方法的匿名值

如果一个嵌入字段带方法，那我们就可以在外部结构体中直接调用它，并且只有嵌入的字段（而不是整个外部结构体）会作为接收者传递给这些方法。

```
type Tasks struct {  
    slice []string          // 具名字段（聚合）  
    Count                // 匿名字段（嵌入）  
}  
  
func (tasks *Tasks) Add(task string) {  
    task.slice = append(task.slice, task)  
    task.Increment()        // 就像写tasks.Count.Increment()一样  
}  
  
func (tasks *Tasks) Tally() int {  
    return int(task.Count)  
}
```

我们前面讲过Count类型。Tasks结构体有两个字段：一个聚合的字符串切片和一个嵌入的Count值。正如Tasks.Add()方法的实现所说明的那样，我们可以直接访问匿名的Count值的方法。

```
tasks := Tasks{}  
fmt.Println(tasks.IsZero(), tasks.Tally(), tasks)  
tasks.Add( " One " )  
tasks.Add( " Two " )  
fmt.Println(tasks.IsZero(), tasks.Tally(), tasks)  
true 0 {[ ] 0}  
false 2 {[One Two] 2}
```

这里我们创建了两个 Tasks 值，并调用了它们的Tasks.Add()、Tasks().Tally()和Tasks.Count.IsZero()（以 Tasks.IsZero()的形式）方法。虽然我们没有定义Tasks.String()方法，但是当要打印Tasks变量的时

候，Go语言仍然能够智能地将其打印出来。（值得注意的是，我们没有把Tally()方法叫做Count()，是因为嵌入的Tasks.Count值与此有冲突，会导致程序无法编译。）

需重点注意的是，当调用嵌入字段的某个方法时，传递给该方法的只是嵌入字段自身。因此，当我们调用Tasks.IsZero()、Tasks.Increment()，或者任何其他在某个Tasks值上调用的Count方法时，这些方法接受到的是一个Count值（或者*Count值），而非Tasks值。

本例中Tasks类型定义了它自己的方法（Add()和Tally()），同时也有嵌入的Count类型的方法（Increment()、Decrement()和IsZero()方法）。当然，也可以让Tasks类型覆盖任何Count类型中的方法，只需以相同的名字实现该方法就行。（前面我们已经看过了一个相关的例子，参见6.2.1.1节）。

嵌入接口

结构体除了可以聚合和嵌入具体的类型外，也可以聚合和嵌入接口。（自然地，反之在接口中聚合或者嵌入结构体是行不通的，因为接口是完全抽象的概念，所以这样的聚合与嵌入毫无意义）。当一个结构体包含聚合（具名的）或者嵌入（匿名的）接口类型的字段时，这意味着该结构体可以将任意满足该接口规格的值存储在该字段中。

让我们以一个简单的例子结束对结构体的讨论，该例子展示了如何让“选项”支持长名字和短名字（例如，“-o”和“-outfile”）且规定选项值为某特定类型（int、float64和string），以及一些通用的方法。（该例子主要用于做说明用，而非为了其优雅性。如果需要一个全功能的选项解析器，可以查看标准库中的flag包，或者godashboard.appspot.com/project上的某个第三方选项解析器。）

```
type Optioner interface {  
    Name() string
```

```

    IsValid() bool
}
type OptionCommon struct {
    ShortName string "short option name"
    LongName string "long option name"
}

```

`Optioner` 接口声明了所有选项类型都必须提供的通用方法。`OptionCommon` 结构体定义了每一个选项常用到的字段。`Go`语言允许我们用字符串（用`Go`语言的术语来说是标签）对结构体的字段进行注释。这些标签并没有什么功能性的作用，但与注释不同的是，它们可以通过`Go`语言的反射支持来访问（参见9.4.9节）。有些程序员使用标签来声明字段验证。例如，对字符串使用像“`check:len(2, 30)`”这样的标签，或者对数字使用“`check:range(0, 500)`”这样的标签，或者使用程序员自定义的任何语义。

```

type IntOption struct {
    OptionCommon           // 匿名字段（嵌入）
    Value, Min, Max int     // 具名字段（聚合）
}

func (option IntOption) Name() string {
    return name(option.ShortName, option.LongName)
}

func (option IntOption) IsValid() bool {
    return option.Min <= option.Value && option.Value <= option.Max
}

func name(shortName, longName string) string {
    if longName == "" {
        return shortName
    }
}

```



```

    }
    return longName
}

```

上面代码片段包括IntOption自定义类型和一个辅助函数name()的完全实现。由于嵌入了OptionCommon结构体，我们可以直接访问它的字段，正如我们在IntOption.Name()方法中所使用的那样。IntOption 满足Optioner 接口（因为它提供了一个 Name()和IsValid()方法，而其签名也一样）。

虽然 name()所做的处理非常简单，我们还是选择将其功能独立出来，而非在IntOption.Name()中实现。这使得IntOption.Name()函数非常简短，并且也让我们可以在其他自定义选项中重用这些功能。因此，像GenericOption.Name()和StringOption.Name()这样的方法其方法体等价于IntOption.Name()中的单语句方法体，而这3条语句都依赖于name()函数完成实质性的工作。这是Go语言中非常普通的模式，我们将在本章的最后一节中再次看到这种模式。

StringOption的实现非常类似于IntOption的实现，因此我们没有给出。（不同点在于，它的Value字段是string类型的，而它的IsValid()方法在Value值为非空的情况下返回true。）对于FloatOption类型，我们使用了嵌入的接口，下面给出它是如何实现的。

```

type FloatOption struct {
    Optioner // 匿名字段（接口嵌入：需要具体的类型）
    Value    float64 // 具名字段（聚合）
}

```

这是 FloatOption 类型的完全实现。嵌入的Optioner 字段意味着当我们创建一个FloatOption值时，必须给该字段赋一个满足该接口的值。

```

type GenericOption struct {

```

```

    OptionCommon // 匿名字段（嵌入）
}
func (option GenericOption) Name() string {
    return name(option.ShortName, option.LongName)
}
func (option GenericOption) IsValid() bool {
    return true
}

```

这是GenericOption类型的完全实现，它满足Optioner接口。

FloatOption 类型有一个嵌入的 Optioner 类型的字段，因此 FloatOption 值需要一个具体的类型来满足该字段的Optioner接口。这可以通过给FloatOption值的Optioner字段赋一个GenericOption类型的值来实现。

现在我们定义了所需的类型（IntOption和FloatOption等），让我们看看如何创建并使用它们。

```

fileOption := StringOption{OptionCommon{ " f " , " file " }, "
index.html " }
topOption := IntOption {
    OptionCommon: OptionCommon{ " t " , " top " },
    Max: 100,
}
sizeOption := FloatOption{
    GenericOption{OptionCommon{ " s " , " size " }}, 19.5}
for _, option := range []Optioner{topOption, fileOption, sizeOption} {
    fmt.Print( " name= " , option.Name(), " •valid= " , option.IsValid())
    fmt.Print( " •value= " )
    switch option := option.(type) { // 影子变量

```

```

    case IntOption:
        fmt.Print(option.Value, " •min= ", option.Min, " •max= ",
        optiuon.Max, " \n ")
    case StringOption:
        fmt.Println(option.Value)
    case FloatOption:
        fmt.Println(option.Value)
}
}

```

```

name=top•valid=true•value=0•min=0•max=100
name=file•valid=true•value=index.html
name=size•valid=true•value=19.5

```

StringOption类型的fileOption值使用传统的方式创建，并且每一个字段都按顺序被赋以一个合适值。但是对于IntOpiton类型的topOption值，我们只为OptionCommon和Max字段赋值，而其他字段只需零值就够了（即Value字段和Min字段只需零值就够了）。Go语言允许我们使用fieldName: fieldValue的形式初始化我们创建的结构体的值中的字段。使用这种语法后，任何没有显式赋值的字段都被自动赋值为零值。

FloatOption类型的sizeOption值的第一个字段是一个Optioner接口，因此我们必须提供一个满足该接口的具体类型。为此，我们在这里创建了一个GenericOption值。

创建了3个不同的选项后我们就可以使用[]Optioner，即一个保存满足Optioner接口的值的切片来迭代它们。在循环中，option变量轮流保存每个选项（其类型为Optioner）。我们可以通过option变量来调用Optioner接口中声明的任何方法，这里我们调用了Option.Name()和Option.IsValid()方法。

每一个选项类型都有一个**Value**字段，但是它们是属于不同类型的。例如，`IntOption.Value`是一个**int**类型，而**`StringOption.Value`**是一个**string**类型。因此，为了访问特定类型的**Value**字段（任何其他特定类型的字段或者方法也类似），我们必须将给定的选项转换为正确的类型。这可以通过使用一个类型开关（参见5.2.2.2节）来轻松完成。在上面的类型开发代码片段中，我们创建了一个影子变量（`option`），它在**case**语句中执行时总是拥有正确的类型（例如，在**`IntOption`** **case**语句中，`option`是**`IntOption`**类型，等等），因此在每个**case**语句中，我们都能够访问任何特定类型的字段或者方法。

6.5 例子

既然我们知道了如何创建自定义类型，就让我们来看一些更为实际和复杂的例子。第一个例子展示了如何创建一个简单的自定义类型。第二个例子展示了如何使用嵌入来创建一系列相关接口和结构体，以及如何提供类型构造函数和创建包中所有导出类型的值的工厂函数。第三个例子展示了如何实现一个完整的自定义通用集合类型。

6.5.1 FuzzyBool——一个单值自定义类型

在本节中，让我们看看如何创建一个基于单值的自定义类型及其支撑方法。这个示例基于一个结构体，保存在文件**`fuzzy/fuzzybool/fuzzybool.go`**中。

内置的布尔类型是双值的（**`true`**和**`false`**），但在一些人工智能领域中，使用的是模糊（**`fuzzy`**）布尔类型。它们的值与“**`true`**”和“**`false`**”相关，并且是介于它们之间的中间体。在我们的实现，我们使用一个浮

点值，0.0表示false而1.0表示true。在这个系统中，0.5表示50%的真（50%的假），而0.25表示0.25%的真(75%的假)，依次类推。这里有些使用示例及其产生的结果。

```
func main() {
    a, _ := fuzzybool.New(0) // 使用时可以安全地忽略err值
    b, _ := fuzzybool.New(.25) // 已确定是合法的值。使用时需确认
    c, _ := fuzzybool.New(.75) // 仍是变量
    d := c.Copy()
    if err := d.Set(1); err != nil {
        fmt.Println(err)
    }
    process(a, b, c, d)
    s := []*fuzzybool.FuzzyBool{a, b, c, d}
    fmt.Println(s)
}

func process(a, b, c, d *fuzzybool.FuzzyBool) {
    fmt.Println(" Original: ", a, b, c, d)
    fmt.Println(" Not: ", a.Not(), b.Not(), c.Not(), d.Not())
    fmt.Println("   Not   Not:   ", a.Not().Not(), b.Not().Not(),
c.Not().Not(),
        d.Not().Not())
    fmt.Print(" 0.And(.25)→ ", a.And(b), " •.25.And(.75)→ ",
b.And(c),
        " •.75.And(1)→ ", c.And(d), " •.25.And(.75,1)→ ", b.And(c,
d), " \n ")
    fmt.Print(" 0.Or(.25)→ ", a.Or(b), " •.25.Or(.75)→ ", b.Or(c),
```

```

    " •.75.Or(1)→ " , c.Or(d), " •.25.Or(.75,1)→ " , b.Or(c, d), " \n
    " )
    fmt.Println( " a < c, a == c, a > c: " , a.Less(c), a.Equal(c), c.Less(a))
    fmt.Println( " Bool: " , a.Bool(), b.Bool(), c.Bool(), d.Bool())
    fmt.Println( " Float: " , a.Float(), b.Float(), c.Float(), d.Float())
}

```

```

Original: 0% 25% 75% 100%
Not: 100% 75% 25% 0%
Not Not: 0% 25% 75% 100%
0.And(.25)→0%.25.And(.75)→25%.75.And(1)→75%
0.And(.25,.75,1)→0%
0.Or(.25)→25%.25.Or(.75)→75%.75.Or(1)→100%
0.Or(.25,.75,1)→100%
a < c, a == c, a > c: true false false
Bool: false false true true
Float: 0 0.25 0.75 1
[0% 25% 75% 100%]

```

该自定义类型叫做 **FuzzyBool**。我们从类型定义开始看起，然后再看其构造函数。最后再看看它的方法定义。

```
type FuzzyBool struct{ value float32 }
```

FuzzyBool类型基于一个包含单**float32**值的结构体。该值是不可导出的，因此任何导入**fuzzybool**包的用户都必须使用构造函数（按照Go语言的惯例，我们将其定义为**New()**）来创建模糊布尔值。当然，这意味着我们可以保证只创建包含合法值的模糊布尔值。

由于**FuzzyBool**类型是基于结构体的，而该结构体所包含的值的类型在结构体中是独一无二的，因此我们可以将其定义简化为**type FuzzyBool struct{ float32 }**。这意味着需要将访问该值的代码从

`fuzzy.value`更改为`fuzzy.float32`，包括下面我们将看到的一些方法中的代码。我们更倾向于使用具名变量，部分是因为这样更为美观，部分是因为如果我们要更改该结构体的底层类型（如改成`float64`），我们只需做少量的更改。

往后的更改也有可能，因为该结构体只包含一个单值。例如，我们可以将其类型更改为`type FuzzyBool float32`，使它直接基于`float32`。这样做能够很好地工作，但稍微需要多点代码，并且与基于结构体的方式相比较，实现起来也稍微麻烦。然而，如果将我们自己局限于创建不可变的模糊布尔值（唯一的区别在于，不是使用`Set()`方法来设置新值，而是直接使用一个新的模糊布尔值赋值），通过直接基于`float32`类型的方式，我们可以极大地简化代码。

```
func New(value interface{}) (*FuzzyBool, error) {  
    amount, err := float32ForValue(value)  
    return &FuzzyBool{amount}, err  
}
```

为了方便模糊布尔值的用户，除了只接受一个 `float32` 值作为初始值之外，我们也可以接受`float64`型（Go语言的默认浮点类型）、`int`型（默认的整型）以及布尔值。这种灵活性是通过使用 `float32ForValue()` 函数来达到的，对应给定的值，它会返回一个 `float32`和`nil`，或者如果的给定值没法处理则返回`0.0`和一个错误值。

如果我们传入了一个非法值，就犯了一个编程错误，我们希望马上知道该错误。但我们并不希望程序在用户那里崩溃。因此，除了返回一个`*FuzzyBool`值外，我们也返回错误值。如果我们给`New()`函数传入一个合法的字面量（正如前文代码片段中所见，），我们可以安全地忽略错误。但是如果我们传入的是一个变量，就必须检查返回的错误值，以防它不是非空值。

`New()`函数返回一个指向**FuzzyBool**类型值的指针而非一个值，因为我们在实现中让模糊布尔值是可更改的。这也意味着这些修改模糊布尔值的方法（本例中只有一个**Set()**）必须接受一个指针接收者，而非一个值 [5]。

一个合理的经验法则是，对于不可变的类型创建只接受值接收者的方法，而为可变的类型创建接受指针接收者的方法。（对于可变类型，让部分方法接受值而让其他方法接受指针是完全可行的，但是在实际使用中可能不太方便。）同时，对于大的结构体类型（例如，那些包含两个或者更多个字段的类型），最好使用指针，这样就能将开销保持在只传递一个指针的程度。

```
func float32ForValue(value interface{}) (fuzzy float32, err error) {
    switch value := value.(type) { // 影子变量
    case float32:
        fuzzy = value
    case float64:
        fuzzy = float32(value)
    case int:
        fuzzy = float32(value)
    case bool:
        fuzzy = 0
    if value {
        fuzzy = 1
    }
    default:
        return 0, fmt.Errorf( " float32ForValue(): %v is not a " +
            " number or Boolean " , value)
    }
}
```



```

    if fuzzy < 0 {
        fuzzy = 0
    } else if fuzzy > 1 {
        fuzzy = 1
    }
    return fuzzy, nil
}

```

该非导出的辅助函数用于在 `New()`和`Set()`方法中将一个值导出为 `[0.0, 1.0]`范围内的`float32`值。通过使用类型开关（参见5.2.2.2节）来处理不同的类型非常简单。

如果该函数以一个非法值调用，我们就返回一个非空值错误。调用者有责任检查返回值并在错误发生时采取相应处理。调用者可以抛出异常以让应用程序崩溃，或者自己来处理问题。出现问题时，这样的底层函数返回错误值是种很好的做法，因为它们没有足够多关于程序逻辑的信息，来了解如何或者是否处理错误，而只是将错误向上推给调用者，而调用者更清楚应该如何处理。

虽然我们将传入非法值当做一种编程错误且认为应该返回一个非空的错误值，我们对超出预期的值采取从简处理，只将其转换成最接近的合法值。

```

func (fuzzy *FuzzyBool) String() string {
    return fmt.Sprintf( " %.0f%% ", 100*fuzzy.value)
}

```

该方法满足 `fmt.Stringer` 接口。这意味着模糊布尔值会按声明的方式输出，而模糊布尔值可以传递给任何接受`fmt.Stringer`值的地方。

我们让模糊布尔值的字符串表示成数字百分比。（回想一下，“%.0f”字符串格式声明了一个没有小数点也没有小数位的浮点类型数

字，而“%%”格式声明了字面量%字母。字符串格式相关的内容在前文已有阐述，参见3.5节。）

```
func (fuzzy *FuzzyBool) Set(value interface{}) (err error) {  
    fuzzy.value, err = float32ForValue(value)  
    return err  
}
```

该方法使得我们的模糊布尔变量变得可更改。该方法与New()函数非常类似，只是这里我们工作于一个已存在的*FuzzyBool，而非创建一个新的。如果返回的错误值非空，那么模糊布尔值就是非法的，因此我们希望调用者检查返回值。

```
func (fuzzy *FuzzyBool) Copy() *FuzzyBool {  
    return &FuzzyBool{fuzzy.value}  
}
```

对于需将自定义类型以指针的形式传来传去的情况，提供Copy()方法会更为方便。这里，我们简单创建了一个新的FuzzyBool值，其值与接收者的值相同，并返回一个指向它的指针。这里不用做任何验证，因为我们知道接收者的值一定是合法的。这里假设原始值使用New()函数创建时其返回的错误值为空，对于后续Set()方法调用也有类似的假设。

```
func (fuzzy *FuzzyBool) Not() *FuzzyBool {  
    return &FuzzyBool{1 - fuzzy.value}  
}
```

这是第一个逻辑运算方法，并且与其他所有方法一样，它也工作于一个*FuzzyBool接收者。

对于该方法我们本可以有3种合理的设计方式。第一种方式是直接更改调用该方法的值而不返回任何东西。另一种方式是修改调用该方法的值并将修改后的值返回，这是标准库中大多数big.Int和big.Rat类型

的方法所采用的方式。这种方式意味着操作可以被链接（例如，`b.Not().Not()`）。这也可以节省内存（因为值被重用而非重新创建），但也容易让我们在忘记了返回值与其自身是同一个值并且已被改过时措手不及。还有一种方式跟我们这里所采取的方式一样：不改变其值本身，但是返回一个新的经过逻辑运算的模糊布尔值。这很容易理解和使用，并且也支持链式，代价是创建了更多值。我们在所有的逻辑运算函数中都使用最后一种方式。

顺便提一下，模糊的“非”逻辑非常简单，对于 1.0 值返回 0.0，对于 0.0 值返回 1.0，对于 0.75 值返回 0.25，对于 0.25 返回 0.75，对于 0.5 值返回 0.5，依次类推。

```
func (fuzzy *FuzzyBool) And(first *FuzzyBool, rest...*FuzzyBool)
    *FuzzyBool {
    minimum := fuzzy.value
    rest = append(rest, first)
    for _, other := range rest {
        if minimum > other.value {
            minimum = other.value
        }
    }
    return &FuzzyBool{minimum}
}
```

模糊的“与”操作的逻辑是返回给定模糊值中最小的那个。该方法的签名保证调用该方法时，调用者至少会传入一个别的 `*FuzzyBool` 值（`first`），另外，还接受零到多个同类型的值（`rest`）。该方法只是简单地将 `first` 值添加进（可能为空的）`rest` 切片的末尾，然后迭代该切片，如果发现 `minimum` 值比迭代过程中的值大，则将 `minimum` 值设为当

前迭代的值。同时，就像 `Not()` 方法一样，我们会返回一个新的 `*FuzzyBool` 值，并将原始的调用方法的模糊布尔值保持不变。

模糊的“或”操作的逻辑是返回给定模糊值中最大的那个。我们没有给出 `Or()` 方法是因为它结构上与 `And()` 方法相同。唯一的区别就是 `Or()` 方法使用一个 `maximum` 变量而非一个 `minimum` 变量，并且比较的时候使用的是 `<` 小于操作符而非 `>` 大于操作符。

```
func (fuzzy *FuzzyBool) Less(other *FuzzyBool) bool {
    return fuzzy.value < other.value
}

func (fuzzy *FuzzyBool) Equal(other *FuzzyBool) bool {
    return fuzzy.value == other.value
}
```

这两个方法允许我们以它们所包含的 `float32` 值的形式比较模糊布尔值。两个方法的返回值都为布尔值。

```
func (fuzzy *FuzzyBool) Bool() bool {
    return fuzzy.value >=.5
}

func (fuzzy *FuzzyBool) Float() float64{
    return float64(fuzzy.value)
}
```

可以将 `fuzzybool.New()` 构造函数看成一个转换函数，因为给定 `float32`、`float64`、`int` 和 `bool` 型的值，它都能够输出一个 `*FuzzyBool` 值。这两个方法采用别的方式进行类似的转换。

`FuzzyBool` 类型提供了一个完整的模糊布尔数据类型，可以像其他所有自定义类型一样使用。因此，`*FuzzyBool` 可以存储在切片中，或者以键或值甚至既是键也是值的形式存储在映射（`map`）中。当然，如果我们使用 `*FuzzyBool` 来做一个映射（`map`）的键值，我们就可以存储

多个模糊布尔值，哪怕它们值是相同的，因为它们每个都含有不同的地址。一种解决方案是采用基于值的模糊布尔值（例如本书源代码中的 `fuzzy_value` 例子）。另一种方法是，我们可以定义自定义集合类型，使用指针来存储，但使用它们的值来进行比较。自定义的 `omap.Map` 类型也能完成这些功能，只要提供一个合适的小于函数（参见6.5.3节）。

除了本节给出的模糊布尔类型外，本书的例子中也包含3个备选的模糊布尔实现供比较。这些备选方案没在本书中给出也未详细讨论。第一个可选的实现在文件 `fuzzy_value/fuzzybool/fuzzybool.go` 和 `fuzzy_mutable/fuzzybool/fuzzybool.go` 中，其功能与本节给出的版本完全一样（在文件 `fuzzy/fuzzybool/fuzzybool.go` 中）。`fuzzy_value` 版本是基于值的，而非 `*FuzzyBool`，而 `fuzzy_mutable` 版本则直接基于一个 `float32` 值而非结构体。`fuzzy_mutable` 的代码稍微比基于结构体的版本冗长而且难懂。第三个可选的版本提供的功能稍微比其他的少，因为它提供的是一个不可变的模糊布尔类型。它也是直接基于 `float32` 类型的，该版本的代码在文件 `fuzzy_immutable/fuzzybool/fuzzybool.go` 中。这是3个可选实现中最简单的一种。

6.5.2 Shapes——一系列自定义类型

当我们希望在一系列相关的类型（例如各种形状）之上应用一些通用的操作时（例如，让一个形状把它们自身画出来），可以采取两种用的比较广泛的实现方法。熟悉C++、Java以及Python的程序员可能会使用层次结构，在Go语言中是嵌套接口。然而，通常更为方便而强大的做法是创建一系列能够相互独立的结构体。在本节中，我们两种方式都会给出，第一种方式在文件 `shaper1/shapes/shapes.go` 中，而第二种方式在文件 `shaper2/shapes/shapes.go` 中。（值得注意的是，由于大多

数包的类型、函数和方法名都是一样的，我们简单地使用“形状包”来指代它们。自然地，当提到具体到某个例子的代码时，我们会以“shaper1形状包”和“shaper2形状包”来区分它们。）

图 6-3 给出了个示例，展示了我们的形状包所能做的事情。这里创建了一个白色的矩形，并在其上画了一个圆，以及一些边数和颜色不一的多边形。

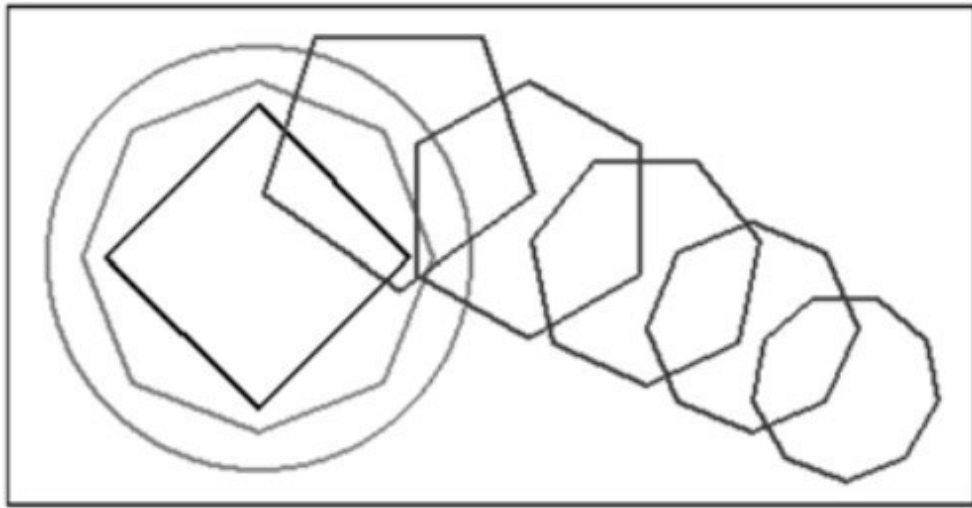


图6-3 shaper示例的shapes.png文件

该形状包提供了3个操作图像的可导出函数，以及3种创建图像的类型，其中两种是可导出的。分层次的shapes1形状包提供了5个可导出接口。我们从图像相关的代码（便捷函数）开始，然后再看看其中的接口（在两个小节中），最后再回顾一下具体形状相关的代码。

6.5.2.1 包级便捷函数

标准库中的image包提供了image.Image接口。该接口声明了3个方法：image.Image.ColorModel()返回图像的颜色模型（以color.Model的形式），image.Image.Bounds()返回图像的边界盒子（以image.Rectangle的形式），而image.Image.At(x, y)返回对应像素的color.Color值。需注意的是，接口image.Image中没有声明设置像素的方法，虽然多个图像类型都提供了Set(x, y int, fill color.Color)方法。不过image/draw包提供了

`draw.Image`接口，它嵌套了`image.Image`接口也包含了一个`Set()`方法。标准库中的`image.Grow`和`image.RGBA`类型以及其他类型都满足`draw.Image`接口。

```
func FilledImage(width, height int, fill color.Color) draw.Image {  
    if fill == nil { // 默认将空的颜色值设为黑色  
        fill = color.Black  
    }  
    width = saneLength(width)  
    height = saneLength(height)  
    img := image.NewRGBA(image.Rect(0, 0, width, height))  
    draw.Draw(img, img.Bounds(), &image.Uniform{fill}, image.ZP,  
draw.Src)  
    return img  
}
```

该导出的便捷函数以给定的规格及统一的填充色创建图像。

函数开始处我们将零值的颜色替换为黑色，并且保证宽度和高度两个维度的值都是合理的。然后创建了一个`image.RGBA`值（一个使用红色、绿色、蓝色以及 α -透明度值创建的图像），并将其以`draw.Image`类型返回，因为我们只关心拿它来做什么，而不关心它的实际值是什么。

`draw.Draw()`函数接受的参数包括一个目标图像（类型为`draw.Image`）、一个声明在哪画图的矩形（在本例中是整个目标图像）、一个用于复制的源图像（本例中是一张以给定颜色填充大小无限的图像）、一个声明模板矩形从哪开始画图的点（`image.ZP`是一个0点，即点(0,0)），以及如何绘制该图的参数。这里，我们声明了`draw.Src`，因此该函数会简单地将原图复制至目标图。因此，我们这里

得到的效果是将给定颜色复制至目标图像中的每一个像素中。（draw包也有一个draw.DrawMask()函数，它支持一些Porter-Daff合成运算。）

```
var saneLength, saneRadius, saneSides func(int) int
func init() {
    saneLength = makeBoundedIntFunc(1, 4096)
    saneRadius = makeBoundedIntFunc(1, 1024)
    saneSides = makeBoundedIntFunc(3, 60)
}
```

我们定义了 3 个未导出的变量来保存辅助函数，这些函数都接受一个 int 值并返回一个int值。同时我们给该包定义了一个init()函数，其中这些变量被赋值成合适的匿名函数。

```
func makeBoundedIntFunc(minimum, maximim int) func(int) int {
    return func(x int) int {
        valid := x
        switch {
        case x < minimum:
            valid = minimum
        case x > maximum:
            valid = maximum
        }
        if valid != x {
            log.Printf( " %s(): replaced %d width %d\n " , caller(1), x,
valid)
        }
        return valid
    }
}
```


该函数返回一个函数。在返回的函数中，对于给定的值`x`，如果它在`minimum`和`maximum`之间（包含这两个值）则返回它，否则返回最接近的边界值。

如果`x`值不合法，除了返回合法的替代值，我们也将相应的问题记录下来。然而，我们并不想报告成在此处创建的函数（即`saneLength()`、`saneRadius()`和`saneSides()`函数）中存在该问题，因为问题属于其调用者。因此，这里我们不记录此处创建函数的名字，而是用一个自定义的`caller()`函数记录了调用者的名字。

```
func caller(steps int) string{
    name := " ? "
    if pc, _, _, ok := runtime.Caller(steps + 1); ok {
        name = filepath.Base(runtime.FuncForPC(pc).Name())
    }
    return name
}
```

`runtime.Caller()`函数返回当前被调用函数的信息，并且也不是在当前`goroutine`中返回。`int`参数定义了往回退多远（即多少层函数）。如果传入的参数值为0，那么只查看当前函数信息（即`shapes.caller()`函数），而如果传入的值为1，则查看该函数的调用者信息，等等。我们加上1以便从函数的调用者开始查看。

函数`runtime.Caller()`能够返回4块信息：程序计数器（我们将其保存在变量`pc`中了）、文件名以及当前调用发生处所在的行（两个都使用空标识符忽略了），以及一个汇报信息是否可以获取到的布尔标识（我们将其保存在`ok`变量中）。

如果成功获取到程序计数器，那么我们就调用`runtime.FuncForPC()`函数以返回一个`*runtime.Func`值，然后在其之上调用`runtime.Func.Name()`方法以获得主调函数的方法名。其返回的名字

像一条路径，例如，对于函数返回/home/mark/goeg/src/shaper1/shapes.FilledRectangle，而对于方法则返回/home/mark/goeg/src/shaper1/shapes.*shape•SetFill。对于小项目而言，该路径没必要，因此我们使用filepath.Base()函数将其剥离掉。然后我们将其名字返回。

例如，如果我们传入一个超界的宽度值和高度值如5000来调用shapes.FilledImage()函数，则saneLength函数会将问题修正。另外，由于存在问题，就会产生一个记录，本例中该记录是“shapes.FilledRectangle(): replaced 5000 with 4096”。之所以产生这样的结果，是因为saneLength()函数使用参数1调用caller()函数，在caller()内部该值被设为2，因此caller()函数会向上回溯3层：它自己（0层）、saneLength()（1层）以及FilledImage()（2层）。

```
func DrawShapes(img draw.Image, x, y int, shapes..Shaper) error {
    for _, shape := range shapes {
        if err := shape.Draw(img, x, y); err != nil {
            return err
        }
    }
    return nil
}
```

这是另一个导出的便捷函数，也是形状包的两种实现中的唯一区别。这里给出的函数来自于层次结构的shapes1形状包。组合型的shapes2形状包区别在于其函数签名中接受的是Drawer值，即满足Drawer接口（它有一个Draw()方法）的值，而非必须包含Draw()、Fill()和SetFill()方法的Shaper类型的值。因此，在本例中，与层次结构的Shaper类型相比，组合的方式意味着我们使用一个更加具体且所需参数

更少的类型（**Drawer**）。我们会在接下来的两个节中讲解这两个接口。

两种情况下函数的函数体及其功能都是一样的。该函数接受一个用于画图的**draw.Image**参数，一个位置参数（以**x**和**y**坐标的形式）以及0个或者更多个**Shaper**(或者**Drawer**)值。在循环里面，调用每一个形状来在给定的位置绘制其自身。**x**和**y**坐标的值在更底层的形状相关的**Draw()**函数中检查，如果它们是非法的，那么我们会得到一个非空的错误值，然后立即将其返回给调用者。

对于图 6-3，我们使用一个该函数的修改版，它会将图形画 3 遍，一遍是在给定的**x**和**y**坐标，另一遍是在往右偏移一个像素的地方，最后一遍是在往下偏移一个像素的地方。这是为了让截图中的边线显得更粗。

```
func SaveImage(img image.Image, filename string) error {
    file, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer file.Close()
    switch strings.ToLower(filepath.Ext(filename)){
    case ".jpg", ".jpeg":
        return jpeg.Encode(file, img, nil)
    case ".png":
        return png.Encode(file, img)
    }
    return fmt.Errorf(" shapes.SaveImage(): '%s' has an unrecognized "
+
        " suffix ", filename)
```

```
}
```

这是最后一个可导出的便捷函数。给定一个满足 `image.Image` 接口的图像（因为该接口嵌套了一个 `image.Image` 接口，它包含了任何满足 `draw.Image` 接口的方法），该函数尝试将图像保存在一个给定名字的文件中。如果 `os.Create()` 调用失败（例如，由于文件名为空或者 I/O 错误），或者其文件名后缀不可识别，或者图像编码失败，那么函数就会返回一个非空的错误值。

在撰写本书时，Go 语言的标准库支持读和写两种格式的图像：`.png`（`Portable Network Graphics`）和 `.jpg`（`Joint Photographic Experts Group`）。支持更多图像格式的包可以从 godashboard.appspot.com/project 获取。`jpeg.Encode()` 函数有一个额外的参数，可用于微调图像是如何存储的，我们传入 `nil` 值表示使用默认的设置。

这些编码器可能引起异常（例如传入一个空的 `image.Image` 时），因此如果我们要使程序能够容错，就得要么在本函数中要么在调用链的上层函数中延迟调用 `recover()`（参见 5.5.1 节）。我们选择不添加这些保护函数，因为测试套件（这里没给出）会调用该函数足够多次来保证这样的编程错误一旦出现就会被触发并且会导致程序终止，因此几乎不会错过任何错误。

基于传入的 `draw.Image` 接口，我们可以声明一些其像素值可以设为任何我们想要的颜色值的图像。同时，使用 `DrawShapes()` 函数我们可以在这种图像上画出图形（满足 `Shaper` 或者 `Drawer` 接口的图形）。我们可以使用 `SaveImage()` 函数将图片保存在磁盘里。有了这些便捷函数后，我们所需要做的就剩下创建接口（例如 `Shaper` 和 `Drawer` 接口等）和具体的类型和方法以满足这些接口了。

6.5.2.2 嵌套接口的层次结构

有传统的面向对象编程背景的程序员可能倾向于使用 Go 语言的嵌套接口的能力来创建具有层次结构的接口。我们将在下一节看到，推荐方式是使用组合。下面是在基于层次结构的shapes1形状包中所使用的接口。

```
type Shaper interface {
    Fill() color.Color
    SetFill(fill color.Color)
    Draw(img draw.Image, x, y int) error
}

type CircularShaper interface {
    Shaper    // Fill(); SetFill(); Draw()
    Radius()  int
    SetRadius(radius int)
}

type RegularPolygonShaper interface {
    CircularShaper // Fill(); SetFill(); Draw(); Radius(); SetRadius()
    Sides() int
    SetSides(sides int)
}
```

我们创建了一个由 3 个接口组成的层次结构（使用嵌套而非继承），这 3 个接口声明了我们希望定义的形状值具备的方法。

Shaper 接口定义了获得和设置类型为 color.Color 的填充色的方法，以及在一个 draw.Image 的给定位置上绘制其自身的方法。CircularShaper 接口嵌套一个匿名的 Shaper，同时为 int 型的半径添加了一个 getter 和 setter 方法。类似地，RegularPolygonShaper 接口嵌套了一个匿名 CircularShaper 接口（因此也是一个 Shaper 类型），并为一系列类型为 int 的边添加了 getter 和 setter 方法。

虽然像这样创建层次结构可能更熟悉，并且也确实能够完成工作，但在 Go 语言中它并不是完成工作的最好方式。这是因为在我们根本没必要使用层次结构的时候它也能将我们锁在层次结构的世界里，我们真正需要的仅仅是声明下这些特定类型的接口支持一些相关接口。下一节我们将看到，这给了我们更多的灵活性。

6.5.2.3 自由组合的相互独立接口

不失一般性，对于这些形状而言，我们最想描述的是它们所能做的事情（绘制、获取或设置填充颜色、获取或设置半径值等）。下面是组合的 shapes2 形状包中的接口。

```
type Shaper interface{
    Drawer // Draw()
    Filler // Fill(); SetFill()
}

type Drawer interface {
    Draw(img draw.Image, x, y int) error
}

type Filler interface {
    Fill() color.Color
    SetFill(fill color.Color)
}

type Radiuser interface {
    Radius() int
    SetRadius(radius int)
}

type Sideser interface {
    Sides() int
    SetSides(sides int)
```

```
}
```

该包的Shaper接口是一个描述形状的便利途径，即声明该形状可以被绘制且可以获取和设置填充色。每一个其他的接口都声明了一个非常具体的行为（将获取和设置算作一个）。

声明许多独立的接口比使用层次结构灵活得多。例如，与使用层次结构相比，我们可以传入更为具体的类型给DrawShapes()函数。同时，因为无需保持层次结构，我们可以更加自由地添加其他接口。当然，正如我们创建Shaper接口时一样，使用这些细粒度的接口让我们可以更容易组合。

这两个版本的形状包接口完全不一样（虽然都有一个Shaper接口，但它们的接口体不一样）。然而，由于接口和具体类型是完全分离且独立的，这些区别并不影响满足它们的任何具体类型的实现。

6.5.2.4 具体类型与方法

这是讲解形状包的最后一节。本节中，我们会讲解满足上面两节中所述接口的具体实现。

```
type shape struct { fill color.Color }

func newShape(fill color.Color) shape {
    if fill == nil { // 默认将空值颜色设置为黑色
        fill = color.Black
    }
    return shape{ fill }
}

func (shape shape) Fill() color.Color { return shape.fill }

func (shape *shape) SetFill(fill color.Color) {
    if fill == nil { // 默认将空值颜色设置为黑色
        fill = color.Black
    }
}
```

```
    shape.fill = fill
}
```

该简单类型是未导出的，因此只能在相同的形状包内访问。这也意味着在包外无法创建该形状的值。

在层次结构的shaper1形状包中，该类型没有满足任何接口，因为它没提供一个Draw()方法。但是在组合类型的shaper2形状包中，它能够满足Filler接口。

正如代码所示，只有Circle类型（我们稍后讲解）直接嵌套了一个shape值。因此，理论上我们可以将 color.Color 值组合进 Circle 类型中，并让该颜色值的getter和setter函数使用*Circle值而非shape值作为接收者，这样就完全不必使用shape类型。然而，我们更希望保持shape类型，因为它允许我们直接基于该shape类型（为了有颜色）而非Circle类型（因为它们没有半径）创建额外形状接口和类型。后面有个练习可以应用这种灵活性。

```
type Circle struct{
    shape
    radius int
}

func NewCircle(fill color.Color, radius int) *Circle {
    return &Circle{newShape(fill), saneRadius(radius)}
}

func (circle *Circle) Radius() int {
    return circle.Radius
}

func (circle *Circle) SetRadius(radius int) {
    circle.radius = saneRadius(radius)
}
```



```

func (circle *Circle) Draw(img draw.Image, x, y int) error {
    //...省略了大约30行代码
}

func (circle *Circle) String() string {
    return fmt.Sprintf( " circle(fill=%v, radius=%d) " , circle.fill,
circle.radius)
}

```

这是 `Circle` 类型的完全实现。虽然我们可以创建具体的 `*Circle` 值，但也可以以接口的形式传递它们，这给我们带来很大的便利性。例如，`DrawShapes()` 函数（参见 6.5.2.1 节）接受 `Shaper`（或者 `Drawer`），而不管其底层具体类型是什么。

在基于层次结构的 `shaper1` 形状包中，该类型满足 `CircularShaper` 和 `Shaper` 接口。在基于组合的 `shaper2` 包中，它满足 `Filler`、`Radiuser`、`Drawer` 和 `Shaper` 接口。在两种情形下，该类型都满足 `fmt.Stringer` 接口。

由于 Go 语言没有构造函数，而我们有未导出字段，因此我们必须提供构造函数以被显式调用。`Circle` 的构造函数是 `NewCircle()`，稍后我们将看到该包还有一个 `New()` 函数可以用于创建该包中任意形状的值。在前文创建 `saneRadius()` 函数时我们就看到，如果传入的整型参数在某个给定的范围内，`saneRadius()` 辅助函数会直接返回该值，否则会返回另一个合理的值。

`Draw()` 方法的代码被省略了（但是在本书附带的源代码中有给出），因为本章所关心的重点是创建自定义的接口以及类型而非图形处理相关的内容。

```

type RegularPolygon struct {
    *Circle
    sides int
}

```

```

func NewRegularPolygon(fill color.Color, radius, sides int)
*RegularPolygon {
    return &RegularPolygon{NewCircle(fill, radius), saneSides(sides)}
}
func (polygon *RegularPolygon) Sides() int {
    return polygon.sides
}
func (polygon *RegularPolygon) SetSides(sides int) {
    polygon.sides = saneSides(sides)
}
func (polygon *RegularPolygon) Draw(img draw.Image, x, y int) error
{
    //...这里省略了大概55行代码，其中包括两个帮助函数...
}
func (polygon *RegularPolygon) String() string {
    return fmt.Sprintf( " polygon(fill=%v, radius=%d, side=%d) ",
        polygon.Fill(), polygon.Radius(), polygon.sides)
}

```

这里是 **RegularPolygon** 类型的完全实现，它提供了常规多边形类型。该类型与**Circle** 类型非常类似，只是它多了一个更为复杂的**Draw()** 方法（其方法体被省略了）。由于**RegularPolygon**嵌套了一个***Circle**，我们使用**NewCircle()** 函数（该函数会处理验证）为该值赋值。**saneSides()**辅助函数类似于**saneRadius()**函数和**saneLength()**函数。

在基于层次结构的 **shaper1** 图形包中，该类型满足 **RegularPolygonShaper**、**CircularShaper**、**Shaper**和**fmt.Stringer**接口。在基于组合的**shaper2**图形包中，它满足**Filler**、**Radiuser**、**Sideser**、**Drawer**、**Shaper**和**fmt.Stringer**接口。

`NewCircle()`函数和`NewRegularPolygon()`函数允许我们创建`*Circle`和`*RegularPolygon`值，同时由于它们的类型满足`Shaper`和其他接口，我们可以以`Shaper`或者它们所满足的其他任何接口类型的值的形式传递。我们可以在这些值上调用任何 `Shaper`方法（即`Fill()`、`SetFill()`和`Draw()`等方法）。同时如果我们希望在一个`Shaper` 值上调用一个非`Shaper`方法，我们可以使用类型断言或者类型开关以将该值转换为某个包含目标方法的接口形式。讲解`showShapeDetails()`函数的时候我们会看一个例子。

不难发现，我们可以创建许多其他的形状类型，有些是在 `shape` 之上创建的，有些则是基于`Circle`或者`RegularPolyon`。此外，有时我们也希望根据运行时环境来创建形状类型，例如，通过使用一个形状名字。为此，我们可以创建一个工厂函数，即一个返回形状类型的函数，其中返回值的类型取决于一个参数。

```
type Option struct {
    Fill    color.Color
    Radius int
}

func New(shape string, option Option) (Shaper, error) {
    sidesForShape := map[string]int{ " triangle " : 3, " square " : 4,
        " pentagon " : 5, " hexagon " : 6, " heptagon " : 7, " octagon " :
8,
        " enneagon " : 9, " nonagon " : 9, " decagon " : 10}
    if sides, found := sidesForShape[shape]; found {
        return NewRegularPolygon(option.Fill, option.Radius, sides), nil
    }
    if shape != " circle " {
        return nil, fmt.Errorf( " shapes.New(): invalid shape '%s' " , shape)
```

```

    }
    return NewCircle(option.Fill, option.Radius), nil
}

```

该工厂函数需两个参数，即所需创建形状的名字和一个自定义的选项值，其中选项值中可以声明可选的特定形状的参数。（使用结构体来创建可以处理多个可选参数的内容已在第5章阐述。参见5.6.1.3节。）该函数返回一个满足Shaper接口的形状以及空的错误值，或者如果给定的形状名非法则返回空值和一个错误值。（回想一下两个形状包中Shaper接口的不同实现，参见 6.5.2.2 节和6.5.2.3 节。）所创建的特殊形状取决于传入的形状字符串参数。这里没必要验证颜色和半径值，因为这些都交由 shapes.shape.SetFill()方法和shapes.SaneRadius()函数处理了，它们最终又被 NewRegularPolygon()和NewCircle()以及类似的关于多边形的方法调用。

```

    polygon := shapes.NewRegularPolygon(color.RGBA{0, 0x7f, 0, 0x7f},
65, 4)

```

```

    showShapeDetails(polygon) ①

```

```

    y = 30

```

```

    for i, radius := range []int{60, 55, 50, 45, 40} {

```

```

        polygon.SetRadius(radius)

```

```

        polygon.SetSides(i+5)

```

```

        x += radius

```

```

        y += height / 8

```

```

        if err := shapes.DrawShapes(img, x, y, polygon); err != nil {

```

```

            fmt.Println(err)

```

```

        }

```

```

    }

```

上面的代码片段给出了图 6-3 中展示的多边形是如何使用 `DrawShapes()` 函数创建的。`showShapeDetails()` 函数 (①) 用于打印任何形状的详细信息。这样做是可能的，因为该函数接受满足 `Shaper` 接口的任意类型的值（即任何我们定义的形状），而非一个具体的形状类型（例如一个 `*Circle` 或者 `*RegularPolygon`）。

由于两个类型包中的 `Shaper` 接口不一样，因此 `showShapeDetails()` 函数的实现也有两种。下面这种是针对基于层次结构的 `shaper1` 的版本。

```
func showShapeDetails(shape shapes.Shaper) {  
    fmt.Print( " fill= ", shape.Fill(), " " ) // 所有图形都有一个填充色  
    if shape, ok := shape.(shapes.CircularShaper); ok { // 影子变量  
        fmt.Print( " radius= ", shape.Radius(), " " )  
        if shape, ok := shape.(shapes.RegularPolygonalShaper); ok { // 影子变量  
            fmt.Print( " sides= ", shape.Sides(), " " )  
        }  
    }  
    fmt.Println()  
}
```

嵌套不是继承

本小节中，`shaper` 例子解释了如何使用结构体嵌套来达到类似于继承的效果。该技术可能对于将 `C++` 或者 `Java` 代码转换成 `Go` 代码的人（或者那些来自于 `C++` 或者 `Java` 背景的 `Go` 程序员）比较有吸引力。然而，虽然这种方法可行，但 `Go` 语言的方式并不是为了模拟继承，而是为了完全避免继承。

根据例子的上下文，这意味着定义相对独立的结构体：

```
type Circle struct {  
    color.Color  
    Radius int  
}
```

```
type RegularPolygon struct {  
    color.Color  
    Radius int  
    Sides int  
}
```

这样做仍然允许我们传递通用的图形值。毕竟，如果两个图形都有能够满足Drawer接口的Draw()方法，那么Circle和RegularPolygon都可以以Drawer值的形式传递。

另一点需要注意的是，我们让所有字段都是导出的，并没有任何验证。这意味着我们必须在使用时验证其字段，而非在它们被设置时。这两种验证的方式都合理，具体哪种更好取决于环境。

本书的shaper3例子使用上面给出的结构体，并且其功能与本小节给出的shaper1和shaper2例子相同。然而，shaper3更有Go语言的味道，它没有嵌套，并且在使用时做了验证。

在shaper1图形包的接口层次结构中，Shaper接口声明了Fill()和SetFill()方法，因此可以立即使用。但是对于其他方法，我们必须先确认它的类型，看看所传入的类型是否满足声明了我们所需调用函数的接口。例如，在这里，只有当该图形满足CircularShaper接口时才能访问Radius()方法，RegularPolygonalShaper接口的Sides()方法也类似。（回想一下，RegularPolygonalShaper嵌套了一个CircularShaper。）

shaper2版本的showShapeDetails()函数类似于shaper1版本。

```
func showShapeDetails(shape shapes.Shaper) {  
    fmt.Print( " fill= ", shape.Fill(), " " ) // 所有图形都有一个填充色  
    if shape, ok := shape.(shapes.Radiuser); ok { // 影子变量  
        fmt.Print( " radius= ", shape.Radius(), " " )  
    }  
    if shape, ok := shape.(shapes.Sideser); ok { // 影子变量  
        fmt.Print( " sides= ", shapes.Sides(), " " )  
    }  
}
```

```
    }  
    fmt.Println()  
}
```

基于组合的shaper2图形包中有一个便捷的Shaper接口，它嵌套了Drawer和Filler接口，因此我们知道所传入的图形有一个Fill()方法。与shaper1层次接口不同的是，这里我们可以使用非常具体的类型断言来访问图形所支持的Radius()和Sides()方法。

如果shape、Circle或者RegularPolygon中添加了新方法或者新字段，我们的代码无需更改就能够继续工作。但是如果我们为其中的任何一个接口添加了新方法，那么我们就必须更新受影响的图形类型来提供相应的方法，否则我们的代码就会被破坏。一个更好的可选方案是创建一个新接口以包含新方法，并将其已有的接口嵌套在里面。这不会破坏任何已有的代码，同时让我们选择是否往已有类型中添加新方法，这取决于我们是否希望它们满足已有接口的同时也满足新接口。

对于接口，我们推荐使用组合而非继承的方式。我们推荐使用Go语言风格来做结构体嵌套，也就是定义相互独立的结构体，而非试图模拟继承。当然，一旦有了足够多的Go语言编程经验，作出这样的决定就是出于技术优势而非移植的便利性或者纯粹是习惯问题。

除了本节给出的shaper1和shaper2示例外，本书的例子中包含了shaper3，它展示了“更纯”的Go语言风格。shaper3版本只有一个接口Drawer，以及独立的Circle和RegularPolygon结构体（见本节的“嵌套不是继承”部分所述）。同时，shaper3使用了图形值而非指针，并且在使用时进行验证。shaper2/shapes/shapes.go文件和shaper3/shapes/shapes.go文件都值得一看，比较一下两种实现方式。

[6.5.3 有序映射——一个通用的集合类型](#)

本章的最后一个例子是一个通用的有序映射类型，它能够像Go语言内置的map类型一样保存“键/值”对，只是每一对按键序存储。该有序映射使用了一个左倾的红黑树，因此速度非常快，其查找的时间复杂度为 $O(\log_2 n)$ 。[6] 通过比较发现，如果其项以有序的方式添加，一个非平衡二叉树的性能可以降级到一个链表的性能（ $O(n)$ ）。平衡树之所以没有这种缺陷，是因为它们在添加和删除节点的时候维持了树的平衡，因此能够保留良好性能。

来自于基于继承的面向对象编程（如C++、Java和Python）背景的程序员更倾向于让有序映射支持小于操作符（<操作），或者是一个签名为Less(other) bool的方法。这很容易通过定义一个声明了该方法的Lesser接口，并为int、string或者MyType这样的类型提供一个实现了这些方法的包装器类型来实现。然而，在Go语言中，正确的实现方式有点不同。

对于我们实现的Go语言有序映射，我们不对键的类型做直接的限制。相反，我们给每一个映射一个“小于”比较函数以支持按键比较。这意味着无论我们的键类型是否支持<操作符都没关系，只要我们能为其提供一个合适的小于比较函数。

在看具体的实现之前，让我们来看一个使用案例，从创建和填充一个有序映射开始。

```
words := []string{ " Puttering " , " About " , " in " , " a " , " Small  
" , " Land " }  
wordForWord := omap.NewCaseFoldedKeyed()  
for _, word := range words {  
    wordForWord.Insert(word, strings.ToUpper(word))  
}
```

我们自定义的有序映射在omap包中，其类型为Map。由于该映射的零值没什么实用的地方，因此要创建一个Map，我们必须使用

`omap.New()`函数，或者其他的`Map`构造函数，如我们这里所使用的`omap.NewCaseFoldedKeyed()`函数。该特殊构造函数创建了一个空`Map`并返回一个指向该字典的指针（即一个`*Map`），其预定义的小于比较函数不区分大小写，按键比较。

每一个“键/值”对都使用`omap.Map.Insert()`方法添加。该方法接受两个`interface{}`值，即一个任意类型的键和一个任意类型的值。（然而，其中的键必须是兼容小于比较函数的类型，因此本例中的键必须是字符串。）如果新元素被成功插入映射中，那么`Insert()`方法返回`true`，否则如果给定的元素的键在映射中已经存在（在这种情况下元素的值会被新元素的值替代，这与内置的`map`类型的做法一样），则返回`false`。

```
wordForWord.Do(func(key, value interface{}){  
    fmt.Printf( " %v → %v\n ", key, value)  
})
```

```
a → A  
About → ABOUT  
in → IN  
Land → LAND  
Puttering → PUTTERING  
Small → SMALL
```

`omap.Map.Do()`方法接受一个签名为 `func(interface{}, interface{})`的函数作为参数，对于按键排序的有序映射的每一个元素都调用该函数，将元素的键和值作为参数传递给该函数。这里我们使用`Do()`方法打印`wordForWord`中的所有键和值。

除了插入元素和对所有元素都调用方法之外，我们也可以查询映射中有多少个元素，查找元素以及删除元素。

```
fmt.Println( " length before deleting: ", wordForWord.Len())  
_, containsSmall := wordForWord.Find( " small " )
```

```

fmt.Println( " contains small: " , containsSmall)
for _, key := range []string{ " big " , " medium " , " small " } {
    fmt.Printf( " %t " , wordForWord.Delete(key))
}
_, containsSmall = wordForWord.Find( " small " )
fmt.Println( " \nlength after deleting: " , wordForWord.Len())
fmt.Println( " contains smail: " , containsSmall)

```

```

length before deleting: 6
contains small: true
false false true length after deleting: 5
contains small: false

```

`omap.Map.Len()`方法返回有序映射中元素的个数。`omap.Map.Find()`方法使用以`interface{}`的形式给定的键值查找元素，如果找到则返回元素的值和`true`，否则返回`nil`和`false`值。`omap.Map.Delete()`方法使用给定的键删除元素并返回`true`，否则如果有序映射中不含该元素则什么也不做并返回`false`。

如果要存储某自定义类型的键，我们可以使用`omap.New()`函数来创建`Map`，并给它提供一个合适的小于比较函数。

例如，这里是一个非常简单的自定义类型的实现。

```

type Point struct{X, Y, int}
func (point Point) String() string {
    return fmt.Sprintf( " (%d, %d) " , point.X, point.Y)
}

```

现在我们就可以创建一个有序映射，存储将`*Point`作为键，以它们与原点之间的距离作为值的元素。

在下面的代码片段中，我们创建了一个空的`Map`，并给它传入了一个小于比较函数用于比较`*Point`键。然后，我们创建了一个`*Point`切

片，并用其中的点来填充映射。最后，我们使用`omap.Map.Do()`方法按键的顺序来打印映射的键和值。

```
distanceForPoint := omap.New(func(a, b interface{}) bool {
     $\alpha$ ,  $\beta$  := a.(*Point), b.(*Point)
    if  $\alpha$ .X !=  $\beta$ .X {
        return  $\alpha$ .X <  $\beta$ .X
    }
    return  $\alpha$ .Y <  $\beta$ .Y
})
points := []*Point{{3, 1}, {1, 2}, {2, 3}, {1, 3}, {3, 2}, {2, 1}, {2, 2}}
for _, point := range points {
    distance := math.Hypot(float64(point.X), float64(point.Y))
    distanceForPoint.Insert(point, distance)
}
distanceForPoint.Do(func(key, value interface{}) {
    fmt.Printf( " %v → %.2v\n " , key, value)
})
```

```
(1, 2) → 2.2
(1, 3) → 3.2
(2, 1) → 2.2
(2, 2) → 2.8
(2, 3) → 3.6
(3, 1) → 3.2
(3, 2) → 3.6
```

回想下第4章中我们提到的，Go语言非常智能，允许我们在创建切片字面量的时候去掉内层的类型名和符号，因此在这里 `points` 切片的

创建是这条语句的缩写：`points := []*Point{&Point{3, 1}, &Point{1, 2}, ...}`。

虽然还没有给出，我们仍然可以像`wordForWord`映射中那样使用`distanceForPoint`映射中的`Delete()`、`Find()`和`Len()`等方法，只是前两个方法必须使用`*Point` 值（因为小于比较操作函数工作在`*Point`上，而非`Point`）。

既然我们知道了如何使用有序映射，接下来就让我们检查下它的具体实现。我们不会阐述`Delete()`方法的辅助方法及函数，因为其中有些函数或者方法非常具有技巧性，而对它们的阐述并不涉及Go语言编程方面的知识。（当然，所有这些函数都可以从本书的源代码中找到，参见文件qtrac.eu/omap/omap.go。）我们首先看看用于实现有序映射的两个类型（`Map`和`node`），再看看一些构造函数。然后，我们会看看`Map`的方法以及相应的辅助函数。在Go语言编程中非常常见的是，大部分方法都非常简短，而将更为复杂的处理交由辅助函数完成。

```
type Map struct {
    root    *node
    less     func(interface{}, interface{}) bool
    length   int
}

type node struct {
    key, value    interface{}
    red           bool
    left, right   *node
}
```

该有序映射使用两个自定义的结构体类型实现。第一个结构体类型是`Map`结构体，它保存着左倾红黑树的根、一个用于比较键的小于比较函数，以及一个长度值用于存储映射中的元素个数。该类型的字段

都是非导出的，并且小于比较函数的初始零值为`nil`，因此直接创建一个`Map`变量会产生一个非法的`Map`。`Map`类型的文档说明了这点，并引导用户使用`omap`包中的构造函数来创建合法的`Map`。

第二个结构体类型是 `node` 结构体，它表示一个单一的“键/值”项。除了它的键和值字段之外，`node`结构体还有3个额外的字段，用于实现树。`red`字段是布尔类型的，用于表示一个节点是“红”（`true`）还是“黑”（`false`），这用于当树的部分需要旋转以保持平衡时。`left` 字段和`right` 字段是`*node`类型的，它们保存着指向节点左子树及右子树的指针（可能为空值`nil`）。

`omap` 包提供了几个构造函数。这里，让我们看一下通用的`omap.New()`函数及几个其他的函数。

```
func New(less func(interface{}, interface{}) bool) *Map {
    return &Map{less: less}
}
```

这是该包中用于创建任何内置或者自定义类型的有序映射的通用函数，因为我们可以提供一个合适的小于比较函数。

```
func NewCaseFoldedKeyed() *Map {
    return &Map{less: func(a, b interface{}) bool {
        return strings.ToLower(a.(string)) < strings.ToLower(b.(string))
    }}
}
```

该构造函数创建了一个空的有序映射，其键为字符串类型，比较大小时大小写不敏感。

```
func NewIntKeyed() *Map {
    return &Map{less: func(a, b interface{}) bool {
        return a.(int) < b.(int)
    }}
}
```

```
}
```

该构造函数创建了一个空的有序映射，其键类型为int型。

omap 包中也有一个 `omap.NewStringKeyed()` 函数，用于创建其键为区分大小写的字符串的有序映射（其实现与 `omap.NewCaseFoldedKeyed()` 几乎完全相同，只是没有调用 `strings.ToLower()`），还有一个 `omap.NewFloat64Keyed()` 函数与 `omap.NewIntKeyed()` 函数一样，只是它使用的是float64型数据作为键而非int类型。

```
func (m *Map) Insert(key, value interface{}) (inserted bool) {  
    m.root, insterted = m.insert(m.root, key, value)  
    m.root.red = false  
    if insterted {  
        m.length++  
    }  
    return inserted  
}
```

该方法在结构上是一个典型的Go语言方法，因为它将大部分工作都交由一个辅助函数完成，在这里是未导出的`insert()`方法。随着元素的插入，树的根可能被改变，这可能是因为树原本为空而现在包含了一个单节点，该节点必为根，或者因为插入元素后为了维持根节点在内的树平衡必须将树旋转。

无论树的根是否改变，`insert()`方法都会返回树的根及一个布尔值。其中，如果插入了新元素，那么布尔值为 `true`，同时将映射的长度加 1。如果布尔值为 `false`，则意味着给定键所对应的新元素已经在映射中了，因此所做的工作就是用给定的新值替换树中元素的当前值，而映射的长度保持不变。（我们不去解释为什么节点被设置成红色或

者黑色，或者为什么需要将它们旋转。这些内容在Robert Sedgewick的论文中有完整的解释，详情参见前面的备注。)

```
func (m *Map) insert(root *node, key, value interface{}) (*node, bool)
{
    inserted := false
    if root == nil { // 键已经在树中的情况也属于这里
        return &node{key: key, value: value, red: true}, true
    }
    if isRed(root.left) && isRed(root.right) {
        colorFlip(root)
    }
    if m.less(key, root.key) {
        root.left, inserted = m.insert(root.left, key, value)
    } else if m.less(root.key, key) {
        root.right, inserted = m.insert(root.right, key, value)
    } else { // 键已经在树中了，因此只需使用新值替换旧值
        root.value = value
    }
    if isRed(root.right) && !isRed(root.left) {
        root = rotateLeft(root)
    }
    if isRed(root.left) && isRed(root.left.left) {
        root = rotateRight(root)
    }
    return root, inserted
}
```

这是一个递归函数，它会遍历整棵树并查找给定键所在的节点，如有必要会将子树旋转来维持树的平衡。当**Insert()**方法调用该方法的时候，传入的**root**是整棵树的根节点（如果树为空则为**nil**），但随后的递归调用的**root**则为子树的根（可能为**nil**）。

如果新键与已存在的键都不相同，那么遍历会到达一个正确的地方插入该新键，而该地方是一个空的叶子。在这点上我们创建并返回一个新的***node** 作为子树，而其叶子为空 **nil**。我们不必显式地初始化新节点的**left**和**right**字段（即它的叶子），因为Go语言会自动地将其设为默认的零值（即**nil**值），因此我们使用结构体的“键:值”语法只初始化那些非零值的字段。

如果新键与已有的某个键相同，我们重用该已存在键的节点，并简单地将其值替换为新值（其做法与内置的**map**类型一样）。这样做的结果是，一个有序映射中的每个项的键都是唯一的。

```
func isRed(root *node) bool { return root != nil && root.red }
```

该简短的辅助函数返回一个给定的节点是否为红，它把空节点当做黑节点。

```
func colorFlip(root *node) {  
    root.red = !root.red  
    if root.left != nil {  
        root.left.red = !root.left.red  
    }  
    if root.right != nil {  
        root.right.red = !root.right.red  
    }  
}
```

该辅助函数倒置给定节点及其非空叶子节点的颜色。

```

func rotateLeft(root *node) *node {
    x := root.right
    root.right = x.left
    x.left = root
    x.red = root.red

    root.red = true
    return x
}

```

```

func rotateRight(root *node) *node {
    x := root.left
    root.left = x.right
    x.right = root
    x.red = root.red

    root.red = true
    return x
}

```

该函数旋转root的子树并保持其子树平衡。

```

func (m *Map) Find(key interface{}) (value interface{}, found bool) {
    root := m.root
    for root != nil {
        if m.less(key, root.key) {
            root = root.left
        } else if m.less(root.key, key) {
            root = root.right
        } else {
            return root.value, true
        }
    }
    return nil, false
}

```

由于该函数的实现比较直接，并且使用的是迭代而非递归，因此没必要创建一个辅助函数。

该Find()方法通过使用less()函数将当前根的键（因为该方法会遍历整棵树）和目标键进行比较以定位目标元素。这通过使用逻辑等于比较 $x = y \Leftrightarrow \neg(x < y \vee y < x)$ 来完成。这种比较对于int、float64、string、自定义的Point类型以及其他许多类型都有效，但不是对所有类型都有

效。如果需要，也可以很容易地扩展`omap.Map`类型来接受一个独立的比较函数。

需注意的是，我们这里使用了具名返回值，但是从来没有显式地为其赋值。当然，它们在`return`语句中被隐式地赋值了。像这样对返回值进行命名对于函数或者方法的文档来说是个有用的补充。例如，这里可以从`Find(key interface{}) (value interface{}, found bool)`签名很明显地了解返回值是什么。但是，如果其签名是 `Find(key interface{}) (interface{}, bool)`，就没那么明显了。

```
func (m *Map) Delete(key interface{}) (deleted bool) {
    if m.root != nil {
        if m.root, deleted = m.remove(m.root, key); m.root != nil {
            m.root.red = false
        }
    }
    if deleted {
        m.length--
    }
    return deleted
}
```

从一个左倾的红黑树中删除一个元素有一定的技巧性，因此我们将其主要工作交由一个未导出的`remove()`方法以及该方法的辅助函数来完成，这里没有给出`remove()`方法也没给出其辅助函数。如果有序映射是空的，或者如果映射中不含给定的键，那么`Delete()`方法就会安全地不执行任何操作并返回`false`。如果该树只包含一个元素，并且该元素就是需要被删除的，那么`*omap.Map`接收者的根会被设置成空值`nil`（并且树也为空）。如果进行了删除操作，我们会返回`true`，同时将映射的长度减1。

顺便提一下，`remove()`方法使用类似于 `Find()`方法中所使用的比较函数来定位所需删除的元素。

```
func (m *Map) Do(function func(interface{}, interface{})){
    do(m.root, function)
}

func do(root *node, function func(interface{}, interface{})) {
    if root != nil {
        do(root.left, function)
        function(root.key, root.value)
        do(root.right, function)
    }
}
```

`Do()`方法及其`do()`辅助函数用于遍历有序表中的所有元素——按键排序，并针对每个元素将其键和值作为参数以调用传入的函数。

```
func (m *Map) Len() int{
    return m.length
}
```

该方法简单地返回映射的长度。前面看到过，其长度会在 `omap.Map.Insert()`方法和 `omap.Map.Delete()`方法中增加或者减少。

这样就完成了对有序映射这个自定义集合类型的阐述，也到了结束面向对象 Go 语言编程讲解的时候。

如果对于某自定义类型而言任何值都是合法的，我们可以简单地创建该类型（例如，使用一个结构图），并将该类型及其字段导出（以大写字母开头）就足够了。（例如，参考标准库中的 `image.Point`和 `image.Rectangle`类型。）

对于需要验证的自定义类型（例如，那些包含一个或者多个字段的基于结构体的类型，并且要求至少一个字段经过验证），Go 语言有

个特定的编程惯例。必须验证的字段设置成不可导出的（以小写字母开始），同时为其提供getter和setter访问方法。

在当其零值为非法值的类型中，我们将相关字段设为不可导出的，并为其提供访问器方法。我们也提到过，零值为非法时，为其提供一个导出的构造函数（通常叫做 **New()**）。该构造函数通常返回一个指向该类型值的指针，其字段都被设置为合法值。

我们可以传递包含导出以及非导出字段的值以及指向该值的指针。当然，如果类型满足一个或者多个接口，当传递接口有用处时，我们也可以以接口的形式传递该值，也就是说，我们关心的只是该值所能完成的功能，而非该值的类型。

很明显，那些来自于基于继承的面向对象编程背景的程序员（如C++、Java或者Python）需调整他们的思考方式。然而，Go语言中鸭子类型和接口的强大和便利性以及不再需要痛苦地维持继承层次结构，使得投入精力学习是非常值得的。如果按照Go语言的方式来进行，Go语言对面向对象编程方式的效果会非常好。

6.6 练习

本章有3个练习。第一个练习涉及创建一个小的自定义类型，其字段必须是经验证的。第二个练习涉及为本章讲到的某个自定义类型添加新功能。第三个练习需要创建一个小的自定义集合类型。前两个练习不难，但是第三个练习非常有挑战。

(1) 创建一个叫做font的包(例如，在文件my_font/fong.go中)。该包的目的是提供表示字体属性的值（例如，字体的属性和大小）。该包中应该有个 **New()** 函数，它接受一个属性值和一个大小值（两个都必须被验证），返回一个*Font（其字段是合法的非导出字段）。同时提

供一些getter和能够验证的setter方法。对于验证，字体的属性名不能为空，其大小必须在5~144个点之间。如果所给定的值是非法的，就为其设置默认的合法值（或者前一个给 setter提供的值），并记录下问题。同时必须提供一个满足fmt.Stringer接口的方法。

这里有个例子演示了如何创建、操作以及使用该包来打印字体。

```
titleFont := font.New( " serif " , 11)
titleFont.SetFamily( " Helvetica " )
titleFont.SetSize(20)
fmt.Println(titleFont)
```

```
{font-family: " Helvetica " ; font-size: 20pt;}
```

该包准备好后，将例子中的font/font_test.go文件复制至my_font目录中，然后运行go test来做一些基本的测试。

文件font/font.go是一个参考答案。整个包大约50行代码。顺便提一下，我们的做法是让String()方法以CSS格式返回字体的细节。这样做可能有点乏味，但是可以直接地将该包扩展至处理所有的CSS字体属性，如weight、style和variant等。

(2) 将整个shaper例子（分层级的shaper1、基于组合的shaper2或者更具Go语言风格的shaper3，包括它们的子目录，随便你喜欢哪个都可以，但是我们更推荐shaper2和shaper3）拷进一个新目录中，例如my_shaper。编辑my_shaper/shaper[123].go文件：删除除了 image 和 shapes 之外所导入的包，删除 main()函数中的所有语句。编辑my_shaper/shapes/shapes.go 文件，添加支持一种叫做 Rectangle的新形状的代码。该形状需有一个起点和长度宽度（所有image.Rectangle类型所提供的），一个填充色以及一个布尔类型值以表示图形是否需要被填充。像添加其他形状一样添加 Rectangle 来声明该类型的API，也就是说，使用非导出的字段和接口（例如，基于层次结构的

RectangularShaper或者基于组合类型的Rectangler和Filleder)，或者不使用接口而使用可导出的字段（Go语言风格）。Draw()方法并不难，特别是如果你使用该图形包中的未导出的drawLine()函数以及draw.Draw()函数的情况下。同时记住更新New()函数以便能够创建矩形，扩展相应的Option类型。

一旦矩形类型添加完后，my_shaper/shaper[123].go 文件中的main()函数创建和保存的图形就如图6-4所示。

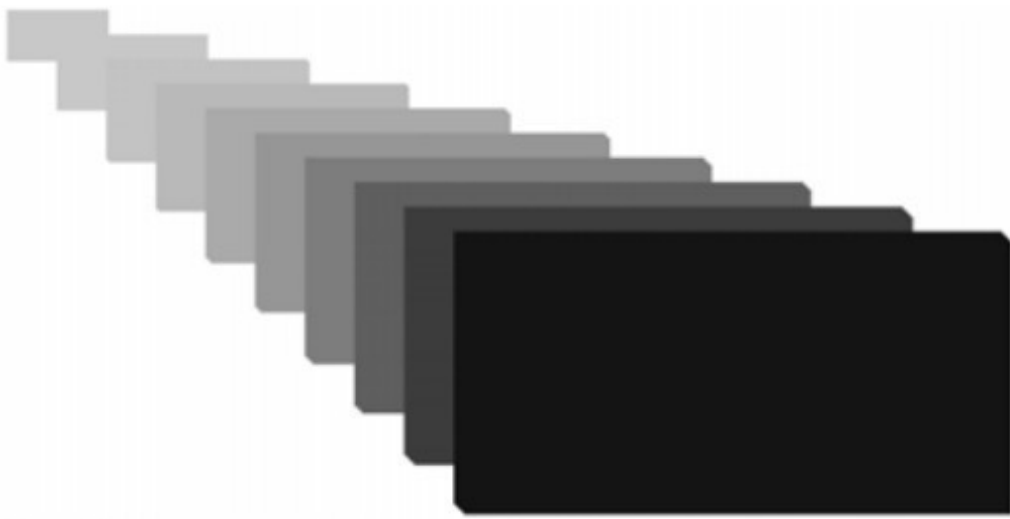


图6-4 一个用矩形类型创建的图形

我们提供了3个参考答案，基于层次结构的实现在shaper_ans1目录下，基于组合类型的实现在shaper_ans2目录下，而更具Go语言风格的实现在shaper_ans3目录下。下面是shaper_ans1方案中用到的RectangularShaper接口：

```
type RectangularShaper interface {  
    Shaper // Fill(); SetFill(); Draw()  
    Rect() image.Rectangle  
    SetRect(image.Rectangle)  
    Filled() bool
```

```
    SetFilled(bool)
}
```

对于shaper_ans2，我们定义了Rectangler和Filleder接口：

```
type Rectangler interface {
    Rect() image.Rectangle
    SetRect(image.Rectangle)
}

type Filleder interface {
    Filled() bool
    SetFilled(bool)
}
```

而更具Go语言风格的解决方案中没有添加新接口。

具体的Rectangle类型本身的代码与基于层次结构和组合结构的代码组织方式相同，并为非导出的字段设置getter和setter方法。但是对于更具Go语言风格的版本，则使用可导出的字段，同时只在使用时进行验证。

```
type Rectangle struct {
    color.Color
    image.Rectangle
    Filled bool
}
```

在文件shaper_ans/shapes/shapes.go中，Rectangle类型及其支持方法一共少于50行代码。Option类型中需多添加几行代码，New()函数中需多添加5行代码。在文件shaper_ans1/shaper1.go中，新的main()函数少于20行代码，和shaper_ans2的实现类似。shaper_ans3实现中额外添加的代码是最少的。

更具创意的读者可能会将该例子展开，为其提供独立的填充色和轮廓色。如果所提供的颜色值为空，则意味着无需绘制，否则使用该颜色进行填充或者勾画轮廓。

(3) 在`my_oslice`包中创建一个名为**Slice**的自定义集合类型。该类型必须实现一个有序的切片，提供几个构造函数，如接受一个小于比较函数的`New(func(interface{}, interface{}) bool)`，以及其他预定义了小于比较函数的构造函数如`NewStringSlice()`和`NewIntSlice()`。除此之外，`*oslice.Slice` 类型还必须实现几个方法，`Clear()`方法用于清空切片，`Add(interface{})`方法用于往切片的恰当位置插入元素，`Remove(interface{}) bool`方法用于移除第一次出现的给定元素并返回是否移除成功，`Index(interface{}) int`方法用于返回给定元素在切片中首次出现的位置（如果不存在返回-1），`At(int)interface{}`方法用于返回给定索引位置下的元素（如果所给的索引位置超出范围则抛出异常），以及一个`Len() int`方法返回切片中元素的个数。

```
func bisectLeft(slice []interface{},
    less func(interface{}, interface{}) bool, x interface{}) int {
    left, right := 0, len(slice)
    for left < right {
        middle := int((left + right) / 2)
        if less(slice[middle], x) {
            left = middle + 1
        } else {
            right = middle
        }
    }
    return left
}
```


`bisectLeft()`函数用于该解决方案中，并且可能非常实用。如果它返回`len(slice)`，则表示给定的元素不在切片中，并且应该放在切片的末尾。任何其他返回值都表示该元素要么在所返回的位置，要么不在切片中而应该放置在所返回的位置中。

有些读者可能会将`oslice/slice_test.go`文件复制到他们的`my_oslice`目录下测试它们的答案。此外，我们还在 `oslice/slice.go` 文件中给出了一个参考答案。`Add()`方法非常具有技巧性，但第 4 章中的`InsertStringSlice()`函数（参见 4.2.3 节）也非常有用。

[1].在Go语言的术语中叫做嵌入的东西在其他某些语言中叫做委托——[delegation](#)。

[2].在C++、Java中接受者统一叫做this，Python中叫做self，在Go语言实践中你可以给它们更有实际意义的名字。

[3].回想一下，在Go语言中，如果标识符以小写字母开头，那么它是非导出的（即只在定义它的包中可见）。如果标识符是以大写字母开头的，那么它是导出的（即在任何导入了定义该标识符的包的包中）。

[4].如果我们刚才是在创建一个框架，我们可能创建一个接口，但是不创建实现该接口的类型，而让使用该框架的用户自己通过创建这些类型来使用框架。

[5].事实上，我们可以返回一个FuzzyBool值，它仍然是一个可变的类型，本书源代码中的fuzzy_value例子有解释。

[6].我们的有序映射是基于左倾的红黑树实现的，关于它的描述请参考Robert Sedgewick的www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf和www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf。本书撰写时，论文中所给出的Java实现是不完整的并且有些错误，因此我们使用Lee Stanaza的C++代码中的思想将其完整地实现了，C++代码请参考www.teachsolaishgames.com/articles/balanced_left_leaning.html。

第7章 并发编程

并发编程可以让开发者实现并行的算法以及编写充分利用多处理器和多核性能的程序。在当前大部分主流的编程语言里，如C、C++、Java等，编写、维护和调试并发程序相比单线程程序而言要困难很多。而且，也不可能总是为了使用多线程而将一个过程切分成更小的粒度来处理。不管怎么说，由于线程本身的性能损耗，多线程编程不一定能够达到我们想要的性能，而且很容易犯错误。

一种解决办法就是完全避免使用线程。例如，可以使用多个进程将重担交给操作系统来处理。但是，这里有个劣势就是，我们必须处理所有进程间通信，通常这比共享内存的并发模型有更多的开销。

Go语言的解决方案有3个优点。首先，Go语言对并发编程提供了上层支持，因此正确处理并发是很容易做到的。其次，用来处理并发的goroutine比线程更加轻量。第三，并发程序的内存管理有时候是非常复杂的，而Go语言提供了自动垃圾回收机制，让程序员的工作轻松很多。

Go语言为并发编程而内置的上层API基于CSP模型（Communicating Sequential Processes）。这就意味着显式锁（以及所有在恰当的时候上锁和解锁所需要关心的东西）都是可以避免的，因为Go语言通过线程安全的通道发送和接受数据以实现同步。这大大地简化了并发程序的编写。还有，通常一个普通的桌面计算机跑十个二十个线程就有点负载过大了，但是同样这台机器却可以轻松地让成百上千甚至过万个goroutine进行资源竞争。Go语言的做法让程序员理解

自己的程序变得更加容易，他们可以从自己希望程序实现什么样的功能来推断，而不是从锁和其他更底层的東西来考虑。

虽然其他大部分语言对非常底层的并发操作（原子级的两数相加、比较、交换等）和其他一些底层的特性例如互斥量都提供了支持，但是在主流语言里，还没有在语言层面像 Go 语言一样直接支持并发操作的（以附加库方式存在的方式并不能算是语言的组成部分）。

除了作为本章主题的Go语言在较高层次上对并发的支持以外，Go 和其他语言一样也提供了对底层功能的支持。在标准库的`sync/atomic`包里提供了最底层的原子操作功能，包括相加、比较和交换操作。这些高级功能是为了支持实现线程安全的同步算法和数据结构而设计的，但是这些并不是给程序员准备的。Go 语言的`sync`包还提供了非常方便的底层并发原语：条件等待和互斥量。这些和其他大多数语言一样属于较高层次的抽象，因此程序员通常必须使用它们。

Go 语言推荐程序员在并发编程时使用语言的上层功能，例如通道和`goroutine`。此外，`sync.Once` 类型可以用来执行一次函数调用，不管程序中调用了多少次，这个函数只会执行一次，还有`sync.WaitGroup`类型提供了一个上层的同步机制，后面我们会看到。

在第5章（5.4节）我们就已经接触过通道和`goroutine`的基本用法，已经讲过的内容不会在这里再讲一遍，但是内容主要还是这些，所以如果能快速复习一遍之前讲过的内容也许会很有帮助。

这一章我们首先对Go语言并发编程的几个关键概念做一个大概的了解，然后还有5个关于并发编程的完整程序作为示例，并展示了 Go 语言中并发编程的范式。第一个例子展示了如何创建一个管道，为了最大化管道的吞吐量，管道里每一部分都各自执行一个独立的`goroutine`。第二个例子展示了怎么将一个工作切分成让固定的若干个`goroutine`去完成，而每部分的输出结果都是独立的。第三个例子展示了如何创建一个线程安全的数据结构，不需要使用锁或者其他底层的原

语。第四个例子使用了3种不同的方法，展示了如何使用固定的若干个goroutine来独立处理其中的一部分工作，并将最终的结果合并在一块。第五个例子展示了如何根据需要来动态创建goroutine并将每个goroutine的工作输出到一个结果集中。

7.1 关键概念

在并发编程里，我们通常想将一个过程切分成几块，然后让每个goroutine各自负责一块工作，除此之外还有main()函数也是由一个单独的goroutine来执行的（为了方便起见，我们将main()函数所在的goroutine称为主goroutine，其他附加创建用来负责处理相应工作的goroutine简称为工作goroutine，以后如果没有特别说明，我们统一沿用这种叫法，虽然本质上它们都是一样的）。每个工作goroutine执行完毕后可以立即将结果输出，或者所有的工作goroutine都完成后再做统一处理。

尽管我们使用Go语言上层的API来处理并发，但仍有必要去避免一些陷阱。例如，其中一个常见的问题就是很可能当程序完成时我们没有得到任何结果。因为当主goroutine退出后，其他的工作goroutine也会自动退出，所以我们必须非常小心地保证所有工作goroutine都完成后才让主goroutine退出。

另一个陷阱就是容易发生死锁，这个问题有一点和第一个陷阱是刚好相反的，就是即使所有的工作已经完成了，但是主goroutine和工作goroutine还存活，这种情况通常是由于工作完成了但是主goroutine无法获得工作goroutine的完成状态。死锁的另一种情况就是，当两个不同的goroutine（或者线程）都锁定了受保护的资源而且同时尝试去获得对方资源的时候，如图7-1所示。也就是说，只有在使用锁的时候才

会出现，所以这种风险一般在其他语言里比较常见，但在Go语言里并不多，因为Go程序可以使用通道来避免使用锁。

为了避免程序提前退出或不能正常退出，常见的做法是让主goroutine在一个done通道上等待，根据接收到的消息来判断工作是否完成了（我们马上就能看到，7.2.2节和7.2.4节也有介绍，也可以使用一个哨兵值作为最后一个结果发送，不过相对其他办法来说这就显得有点拙劣了）。

另外一种避免这些陷阱的办法就是使用sync.WaitGroup来让每个工作goroutine报告自己的完成状态。但是，使用sync.WaitGroup本身也会产生死锁，特别是当所有工作goroutine都处于锁定状态的时候（等待接受通道的数据）调用 sync.WaitGroup.Wait()。后面我们会看到如何使用sync.WaitGroup（参见7.2.5节）。

就算只使用通道，在Go语言里仍然可能发生死锁。举个例子，假如我们有若干个goroutine可以相互通知对方去执行某个函数（向对方发一个请求），现在，如果这些被请求执行的函数中有一个函数向执行它的goroutine发送了一些东西，例如数据，死锁就发生了。如图7-2所示（后面我们还会看到这种死锁的另一种情况）。

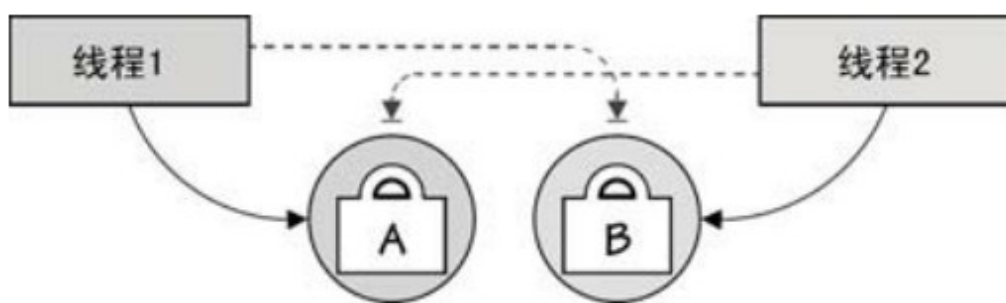


图7-1 死锁：两个或多个阻塞线程试图取得对方的锁

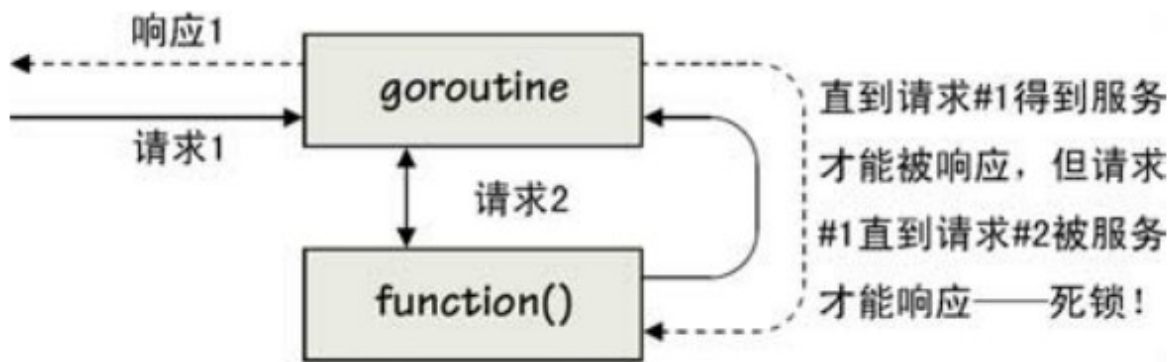


图7-2 死锁：一个试图用对自身的请求来服务于请求的goroutine

通道为并发运行的goroutine之间提供了一种无锁通信方式（尽管实现内部可能使用了锁，但无需我们关心）。当一个通道发生通信时，发送通道和接受通道（包括它们对应的goroutine）都处于同步状态。

默认情况下，通道是双向的，也就是说，既可以往里面发送数据也可以从里面接收数据。但是我们经常将一个通道作为参数进行传递而只希望对方是单向使用的，要么只让它发送数据，要么只让它接收数据，这个时候我们可以指定通道的方向。例如，`chan<- Type` 类型就是一个只发送数据的通道。我们之前的章节里并没有这样用过，一来没这个必要，用 `chan Type` 就行，二来还有很多其他的東西要学习。但是从现在开始，我们会在所有合适的地方使用单向的通道，因为它们会提供额外的编译期检查，这是非常好的处理方式。

本质上说，在通道里传输布尔类型、整型或者 `float64` 类型的值都是安全的，因为它们都是通过复制的方式来传送的，所以在并发时如果不小心大家都访问了一个相同的值，这也没有什么风险。同样，发送字符串也是安全的，因为Go语言里不允许修改字符串。

但是Go语言并不保证在通道里发送指针或者引用类型（如切片或映射）的安全性，因为指针指向的内容或者所引用的值可能在对方接收到时已被发送方修改。所以，当涉及指针和引用时，我们必须保证这些值在任何时候只能被一个goroutine访问得到，也就是说，对这些值

的访问必须是串行进行的。除非文档特别声明传递这个指针是安全的，比如，`*regexp.Regexp` 可以同时被多个 `goroutine` 访问，因为这个指针指向的值的的所有方法都不会修改这个值的状态。

除了使用互斥量实现串行化访问，另一种办法就是设定一个规则，一旦指针或者引用发送之后发送方就不会再访问它，然后让接收者来访问和释放指针或者引用指向的值。如果双方都有发送指针或者引用的话，那就发送方和接受方都要应用这种机制（我们会在7.2.4.3节看到一个使用这个机制的例子），这种方法的问题就是使用者必须足够自律。第三种安全传输指针和引用的方法就是让所有导出的方法不能修改其值，所有可修改值的方法都不引出。这样外部可以通过引出的这些方法进行并发访问，但是内部实现只允许一个 `goroutine` 去访问它的非导出方法（例如在它们的包里，包将会在第9章介绍）。

Go语言里还可以传送接口类型的值，也就是说，只要这个值实现了接口定义的所有方法，就可以以这个接口的方式在通道里传输。只读型接口的值可以在任意多个 `goroutine` 里使用（除非文档特别声明），但是对于某些值，它虽然实现了这个接口的方法，但是某些方法也修改了这个值本身的状态，就必须和指针一样处理，让它的访问串行化。

举个例子，如果我们使用 `image.NewRGBA()` 函数来创建一个新的图片，我们得到一个 `*image.RGBA` 类型的值。这个类型实现了 `image.Image` 接口定义的所有方法（只有一个读取的方法，理所当然只是只读型的接口）和 `draw.Image` 接口（这个接口除了实现了 `image.Image` 接口的所有方法之外，还实现了一个 `Set()` 方法）。所以如果我们只是让某个函数去访问一个 `image.Image` 值的话，我们可以将这个 `*image.RGBA` 值随意发送给多个 `goroutine`。（不幸的是，这种安全性是随时可以颠覆的，比如说，接受方可以使用一个类型断言将这个值转换成 `draw.Image` 接口类型，因此，就必须要有有一种机制能防止这种事

情的发生。) 或者我们希望在多个 `goroutine` 里访问甚至修改同一个 `*image.RGBA` 的值, 就应该以 `*image.RGBA` 或者 `draw.Image` 类型来传送, 不管哪种方式, 都必须让这个值的访问是串行的。

使用并发的最简单的一种方式就是用一个 `goroutine` 来准备工作, 然后让另一个 `goroutine` 来执行处理, 让主 `goroutine` 和一些通道来安排一切事情。例如, 下面的代码是如何在主 `goroutine` 里创建一个名为 “jobs” 的通道和一个叫 “done” 的通道。

```
jobs := make(chan Job)
done := make(chan bool, len(jobList))
```

这里我们创建了一个没有缓冲区的 `jobs` 通道, 用来传递一些自定义 `Job` 类型的值。我们还创建了一个 `done` 通道, 它的缓冲区大小和任务列表的数量是相对应的, 任务列表 `jobList` 是 `[]Job` 类型 (它的初始化这里我们没有列出来)。

只要设置了通道和任务列表 (`jobList`), 我们就可以开始干活了。

```
go func() {
    for _, job := range jobList {
        jobs <- job // 阻塞, 等待接收
    }
    close(jobs)
}()
```

这段代码创建了一个 `goroutine` (`goroutine#1`), 它遍历 `jobList` 切片然后将每一个工作发送到 `jobs` 通道。因为通道是没有缓冲的, 所以 `goroutine#1` 会马上阻塞, 直到有别的 `goroutine` 从这个通道里将任务读取出去。发送完所有任务之后, 就关闭 `jobs` 通道, 这样接收工作的 `goroutine` 就会知道什么时候没有其他工作了。

这段代码所表达的语义还不止这些。例如, `goroutine#1` 会直到 `for` 循环结束才关闭 `jobs` 通道, 而且 `goroutine#1` 会和创建它的主 `goroutine` 并

发执行。还有，`go`声明语句会立即返回，主 `goroutine`（`main()`函数所在的 `goroutine`）继续执行后面的代码，但是由于 `jobs` 通道是没有缓冲的，所以 `goroutine#1` 会反复执行这样一个过程：往 `jobs` 里发送一个任务，等待任务被接收，继续往 `jobs` 里发送任务，等到任务被接收.....直到 `jobList` 任务列表里的所有任务都被处理完后，关闭 `jobs` 通道。显然，从 `for` 循环开始执行到关闭 `jobs` 之间得耗一段时间。

```
go func() {  
    for job := range jobs {    // 等待数据发送  
        fmt.Println(job)      // 完成一项工作  
        done <- true  
    }  
}()
```

这是我们创建的第二个 `goroutine`（`goroutine#2`），遍历 `jobs` 通道，并处理（这里就是打印出来），然后将完成状态 `true`（其实什么都可以，因为我们这里只关心往 `done` 里发了多少个值，而不是实际的什么值）发送到 `done` 通道。

同理，这个 `go` 语句也是立即返回的，`for` 循环会阻塞直到有其他的 `goroutine`（在我们这个例子里，发送数据的是 `goroutine#1`）往通道里发送了数据。此时，整个进程共有 3 个并发的 `goroutine` 在运行，主 `goroutine`、`goroutine#1` 和 `goroutine#2`，如图 7-3 所示。

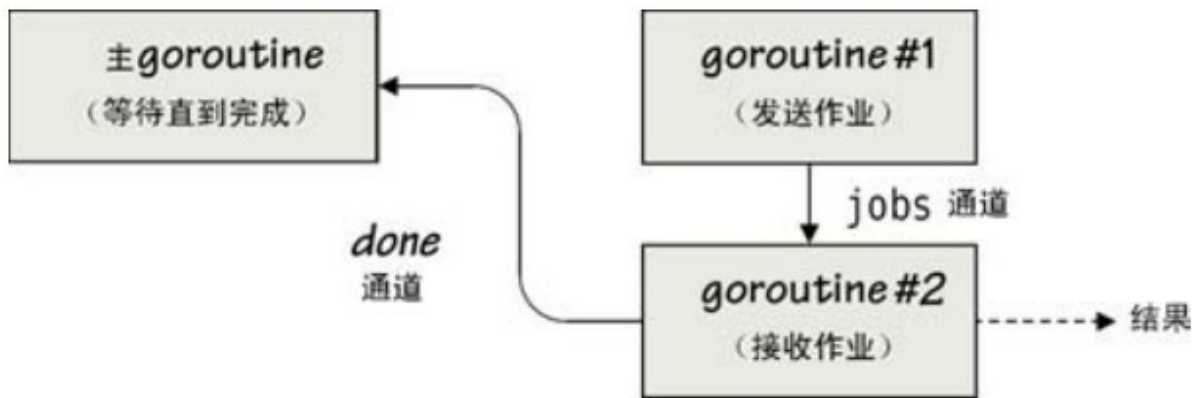


图7-3 并发独立的准备与处理

当goroutine#1发送了一个任务然后等待的时候，goroutine#2就直接接收过来然后处理，期间goroutine#1仍然阻塞，一直持续到它发送第二个工作。一旦goroutine#2处理完一个任务，它就往done通道里发送一个true值。done通道是有缓冲的，所以这个发送操作不会被阻塞。控制流回到goroutine#2的for循环里，它接收下一个从goroutine#1发送过来的工作，如此反复，直到完成所有的工作。

```
for i := 0; i < len(jobList); i++ {  
    <-done // 阻塞，等待接收  
}
```

主goroutine创建完两个工作goroutine后（注意不会阻塞的，所有的goroutine并发执行），继续执行最后一段代码，这段代码的目的是确保主goroutine等到所有的工作完成了才退出。

for循环迭代的次数和任务列表的大小是一样的。每次迭代都从done通道里接收一个值。每次迭代和处理工作都是同步的，只有一个工作被完成后done通道里才有一个值可以被接收（接收到的值将被抛弃）。所有工作完成后，done通道发送和接收数据的次数将和迭代的次数一致，此时for循环也将结束。这时候主goroutine就可以退出了，这样我们也就保证了所有任务都能处理完毕。

对于通道的使用，我们有两个经验。第一，我们只有在后面要检查通道是否关闭（例如在一个`for...range`循环里，或者`select`，或者使用`<-`操作符来检查是否可以接收等）的时候才需要显式地关闭通道；第二，应该由发送端的`goroutine` 关闭通道，而不是由接收端的`goroutine` 来完成。如果通道并不需要检查是否被关闭，那么不关闭这些通道并没有什么问题，因为通道非常轻量，因此它们不会像打开文件不关闭那样耗尽系统资源。

例如我们这个例子就是用`for...range`循环来迭代读取`jobs`通道的，所以我们在发送端把它给关闭了。这和我们的经验是一致的。另外，我们没必要关闭 `done` 通道，因为它后面并没有用在什么特别的语句里（`for...range`或者`select`等）。

这个例子所展示的是 Go语言并发编程里很典型的一种模式，虽然实际上这种情况下这么做没有什么好处。下一章还有一些例子和这个模式是差不多的，但是却非常适合使用并发。

7.2 例子

虽然Go语言里使用`goroutine`和通道的语法很简单，如`<-`、`chan`、`go`、`select`等，但足以应付大多数的并发场合。由于篇幅有限，本章我们无法对所有的并发编程方法都一一介绍，所以这里我们只介绍并发编程中比较常见的3种模式，分别是管道、多个独立的并发任务（需要或者不需要同步的结果）以及多个相互依赖的并发任务，然后我们看下它们如何使用 Go语言的并发支持来实现。

接下来的例子以及本章的练习对 Go语言并发编程实践进行了深入的探讨。你可以将这些实践应用到其他新程序中。

7.2.1 过滤器

第一个例子用于显示一种特定并发编程范式。这个程序可以轻松地扩展以完成更多其他可以从并发模型中获益的任务。

有Unix背景的人会很容易从Go语言的通道回忆起Unix里的管道，唯一不同的是Go语言的通道为双向而Unix管道为单向。利用管道我们可以创建一个连续管道，让一个程序的输出作为另一个程序的输入，而另一个程序的输出还可以作为其他程序的输入，等等。例如，我们可以使用Unix管道命令从Go源码目录树里得到一个Go文件列表（去除所有测试文件）：

```
find $GOROOT/src -name "*.go" | grep -v test.go
```

这种方法的一个妙处就是可以非常容易地扩展。比如说，我们可以增加| `xargs wc -l`来列出每一个文件和它包含的行数，还可以用| `sort -n`得到一个按行数进行排序的文件列表。

真正的Unix风格的管道可以使用标准库里的`io.Pipe()`函数来创建，例如Go语言标准库里就用管道来比较两个图像（在`go/src/pkg/image/png/reader_test.go`文件里）。除此之外，我们还可以利用Go语言的通道来创建一个Unix风格的管道，这个例子就用到了这种技术。

`filter` 程序（源文件是 `filter/filter.go`）从命令行读取一些参数（例如，指定文件大小的最大值最小值，以及只处理的文件后缀等）和一个文件列表，然后将符合要求的文件名输出，`main()`函数的主要代码如下。

```
minSize, maxSize, suffixes, files := handleCommandLine()
sink(filterSize(minSize,      maxSize,      filterSuffixes(suffixes,
source(files))))
```

`handleCommandLine()`函数（这里我们未显示相关代码）用到了标准库里的`flag`包来处理命令行参数。第二行代码展示了一条管道，从最里面的函数调用（`source(files)`开始）到最外面的（`sink()`函数），为了方便大家理解，我们将管道展开如下。

```
channel1 := source(files)
channel2 := filterSuffixes(suffixes, channel1)
channel3 := filterSize(minSize, maxSize, channel2)
sink(channel3)
```

传给`source()`函数的`files`是一个保存文件名的切片，然后得到一个`chan string`类型的通道`channel1`。在`source()`函数中`files`里的文件名会轮流被发送到`channel1`。另外两个过滤函数都是传入过滤条件和`chan string`通道，并各自返回它们自己的`chan string`通道。其中第一个过滤器返回的通道被赋值到`channel2`，第二个被赋值到`channel3`。每个过滤器都会迭代读传入的通道，如果符合条件，就将结果发送到输出通道（这个通道会被返回并可能会作为下一个过滤器的输入源）。`sink()`函数会提取`channel3`里的每一项并打印出来。



图7-4 并发goroutine之间的管道

图7-4简略地阐明了整个`filter`程序发生了什么事情，`sink()`函数是在主`goroutine`里执行的，而另外几个管道函数（如`source()`、`filterSuffixes()`和`filterSize()`函数）都会创建各自的`goroutine`来处理自己的工作。也就是说，主`goroutine`的执行过程会很快地执行到`sink()`这里，此时所有的`goroutine`都是并发执行的，它们要么在等待发送数据要么在等待接收数据，直到所有的文件处理完毕。

```
func source(files []string) <-chan string {  
    out := make(chan string, 1000)  
    go func() {  
        for _, filename := range files {  
            out <- filename  
        }  
        close(out)  
    }()  
    return out  
}
```

`source()`函数创建了一个带有缓冲区的`out`通道用来传输文件名，因为实际测试中缓冲区可以提高吞吐量（这就是我们常说的以空间换时间）。

当输出通道创建完毕后，我们创建了一个`goroutine`来遍历文件列表并将每一个文件名发送到输出通道。当所有的文件发送完毕之后我们将这个通道关闭。`go`语句会立即返回，而且从发送第一个文件名到发送最后一个文件名还有最终关闭通道，这之间可能会有相当长的一个时间差。往通道里发送数据是不会阻塞的（至少，发送前1000个文件时是不会阻塞的，或者说文件数小于1000时不会阻塞），但是如果要发送更多的东西，还是会阻塞的，直到至少通道里有一个数据被接收。

之前我们提到，默认情况下通道是双向的，但我们可将一个通道限制为单向。回忆下前一节我们讲过的，`chan<- Type`是一个只允许发送的通道，而`<-chan Type`是一个只允许接收的通道。函数最后返回的`out`通道就被强制设置成了单向，我们可以从里面接收文件名。当然，直接返回一个双向的通道也是可以的，但我们这里这么做是为了更好地表达程序的思想。

go语句之后，这个新创建的goroutine就开始执行匿名函数里的工作里，它会往out通道里发送文件名，而当前的函数也会立即将out通道返回。所以，一旦调用source()函数就会执行两个goroutine，分别是主goroutine和在source()函数里创建的那个工作goroutine。

```
func filterSuffixes(suffixes []string, in <-chan string) <-chan string {
    out := make(chan string, cap(in))
    go func() {
        for filename := range in {
            if len(suffixes) == 0 {
                out <- filename
                continue
            }
            ext := strings.ToLower(filepath.Ext(filename))
            for _, suffix := range suffixes {
                if ext == suffix {
                    out <- filename
                    break
                }
            }
        }
        close(out)
    }()
    return out
}
```

这是两个过滤函数中的第一个。第二个函数filterSize()的代码也是类似的，所以这里就不显示了。

其实参数里的in 通道是只读或者可读写都是没有关系的，不过，这里我们在参数类型声明时指定了in是一个只读的通道（我们知道source()函数返回的，也就是这个in通道，实际上就是一个只读的通道）。对应地，函数最后将双向（创建时默认就是可读写的）out通道以只读的方式返回，和之前source()的做法一样。其实就算我们忽略掉所有的<-，函数也一样可以工作，但是指定了通道的方向有助于精确地表达到底我们想让程序做什么事情，并借助编译器来强制程序按照这种语义来执行。

filterSuffixes()函数首先创建一个带有缓冲区的输出通道，通道缓冲区和输入通道 in的大小是一样的，以最大化吞吐量。然后程序新建一个goroutine做相应的处理。在goroutine里，遍历in通道（例如，轮流接收每个文件名）。如果没有指定任何后缀的话则任意后缀的文件名我们都接收，也就是简单地发送到输出通道里去。如果我们指定了文件名的后缀，那么只有匹配的文件名（大小写不敏感）才会发送到输出通道，其他的则被丢弃。（filepath.Ext()函数返回文件名的扩展名，也就是它的后缀，包括前导的句点，如果没有匹配的话就返回一个空的字符串。）

和source()函数一样，一旦所有的处理完毕，输出通道就会被关闭，尽管还需要一些时间才会执行到这里。创建goroutine之后输出通道就被函数返回了，这样管道就能从这里接收文件名。

这时，有3个goroutine会在运行，它们是主goroutine和source()函数里的goroutine，以及这个函数里的goroutine。filterSize()函数调用之后就会有4个goroutine，它们都会并发地执行。

```
func sink(in <-chan string) {  
    for filename := range in {  
        fmt.Println(filename)  
    }  
}
```



```
}
```

`source()`函数和两个过滤函数分别在它们各自的goroutine里并发处理，并通过通道来进行通信。`sink()`函数在主goroutine里处理其它函数返回的最后一个通道，它迭代读取成功通过所有过滤器的文件名并进行相应输出。

`sink()`函数的`range`语句遍历一个只读通道，将文件名打印出来或者等待通道被关闭，这样就可以保证主goroutine在所有工作goroutine处理完毕之前不会提前退出。

自然地，我们可以给管道增加一些额外的函数，例如过滤文件名或者处理到目前为止所有通过了过滤器的文件。只要这个函数能接收一个输入通道（前一个函数的输出通道）和返回它自己的输出通道。当然，如果我们想传一些更复杂的值，我们也可以让通道传输的是一个结构而不是一个简单的字符串。

虽然这一节里的管道程序是一个管道框架非常好的示例，不过由于每一阶段处理的东西并不多，所以从管道方案并没有得到非常大的好处。真正能够从并发中获益的管道类型是每一个阶段可能有很多的工作需要处理，或者依赖于别的其他正在被处理的项，这样每个goroutine都能尽可能充分地利用时间。

7.2.2 并发的Grep

并发编程的一种常见的方式就是我们有很多工作需要处理，且每个工作都可以独立地完成。例如，Go语言标准库里的`net/http`包的HTTP服务器利用这种模式来处理并发，每一个请求都在一个独立的goroutine里处理，和其他的goroutine之间没有任何通信。这一节我们以实现一个`cgrep`程序为例说明实现这种模式的一种方法，`cgrep`表示“并发的grep”。

和标准库里的HTTP服务不同的是，**cgrep**使用固定数量的goroutine来处理任务，而不是动态地根据需求来创建。（我们会在后面的7.2.5节看到一个动态创建goroutine的例子。）

cgrep程序从命令行读取一个正则表达式和一个文件列表，然后输出文件名、行号，和每个文件里所有匹配这个表达式的行。没匹配的话就什么也不输出。

cgrep1程序（在文件**cgrep1/cgrep.go**里）使用了3个通道，其中两个是用来发送和接收结构体的。

```
type Job struct {  
    filename string  
    results chan<- Result  
}
```

我们用这个结构体来指定每一个工作，**filename**表示需要被处理的文件，**results**是一个通道，所有处理完的文件都会被发送到这里。我们可以将**results**定义为一个**chan Result**类型，但我们只往通道里发送数据，不会从里面读取数据，所以我们指定这是一个单向的只允许发送的通道。

```
type Result struct {  
    filename  string  
    lino      int  
    line      string  
}
```

每个处理结果都是一个**Result**类型的结构体，包含文件名、行号码，以及匹配的行。

```
func main() {  
    runtime.GOMAXPROCS(runtime.NumCPU()) // 使用所有的机器  
    核心
```

```

    if len(os.Args) < 3 || os.Args[1] == "-h" || os.Args[1] == "--help" {
        fmt.Printf("usage:                %s                <regexp>
<files>\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    if lineRx, err := regexp.Compile(os.Args[1]); err != nil {
        log.Fatalf("invalid regexp: %s\n", err)
    } else {
        grep(lineRx, commandLineFiles(os.Args[2:]))
    }
}

```

程序的main()函数的第一条语句告诉 Go 运行时系统尽可能多地利用所有的处理器，调用 runtime.GOMAXPROCS(0)仅仅是返回当前处理器的数量，但如果传入一个正整数就会设置Go运行时系统可以使用的处理器数。runtime.NumCPU()函数返回当前机器的逻辑处理器或者核心的数量 [1]，Go语言里大多数并发程序的开始处都有这一行代码，但这行代码最终将会是多余的，因为Go语言的运行时系统会变得足够聪明以自动适配它所运行的机器。

main()函数处理命令行参数（一个正则表达式和一个文件列表），然后调用 grep() 函数来进行相应处理（我们在4.4.2节里已经看过 commandLineFiles()函数）。

lineRx是一个*regexp.Regexp类型（参见3.6.5节）的变量，传给 grep()函数并被所有的工作goroutine共享。这里有一点需要注意的，通常，我们必须假设任何共享指针指向的值都不是线程安全的。这种情况下我们必须自己来保证数据的安全性，如使用互斥量（mutex）等。或者，我们为每个工作goroutine单独提供一个值而不是共享它，这就需要多一点内存的开销。幸运的是，对于 *regexp.Regexp，Go语言的文

档说这个指针指向的值是线程安全的，这就意味着我们可以在多个goroutine里共享使用这个指针。

```
var workers = runtime.NumCPU()
func grep(lineRx *regexp.Regexp, filenames []string) {
    jobs := make(chan Job, workers)
    results := make(chan Result, minimum(1000, len(filenames)))
    done := make(chan struct{}, workers)
    go addJobs(jobs, filenames, results)    // 在自己的goroutine中执行
    for i := 0; i < workers; i++ {
        go doJobs(done, lineRx, jobs)      // 每一个都在自己的
        goroutine中执行
    }
    go awaitCompletion(done, results)      // 在自己的goroutine中执行
    processResults(results)                // 阻塞，直到工作完成
}
```

这个函数为程序创建了 3 个带有缓冲区的双向通道，所有的工作都会分发给工作goroutine来处理。goroutine的总数量和当前机器的处理器数相当，jobs通道和done通道的缓冲区大小也和机器的处理器数量一样，将不必要的阻塞尽可能地降到最低。（当然，我们也可以不用管实际机器的处理器数量，而让用户在命令行指定到底需要开启多少个工作goroutine。）对于 results 通道我们像前一小节的filter 程序那样使用了一个更大的缓冲区，然后使用一个自定义的minimum()函数（这里不显示，参见5.6.1.2节的实现，或者cgrep.go源码）。

和之前章节的做法不同，之前通道的类型是chan bool而且只关心是否发送了东西，不关心是true还是false，我们这里的通道类型是chan

`struct{}`（一个空结构），这样可以更加清晰地表达我们的语义。我们能往通道里发送的是一个空的结构（`struct{}{}`），这样只是指定了一个发送操作，至于发送的值我们不关心。

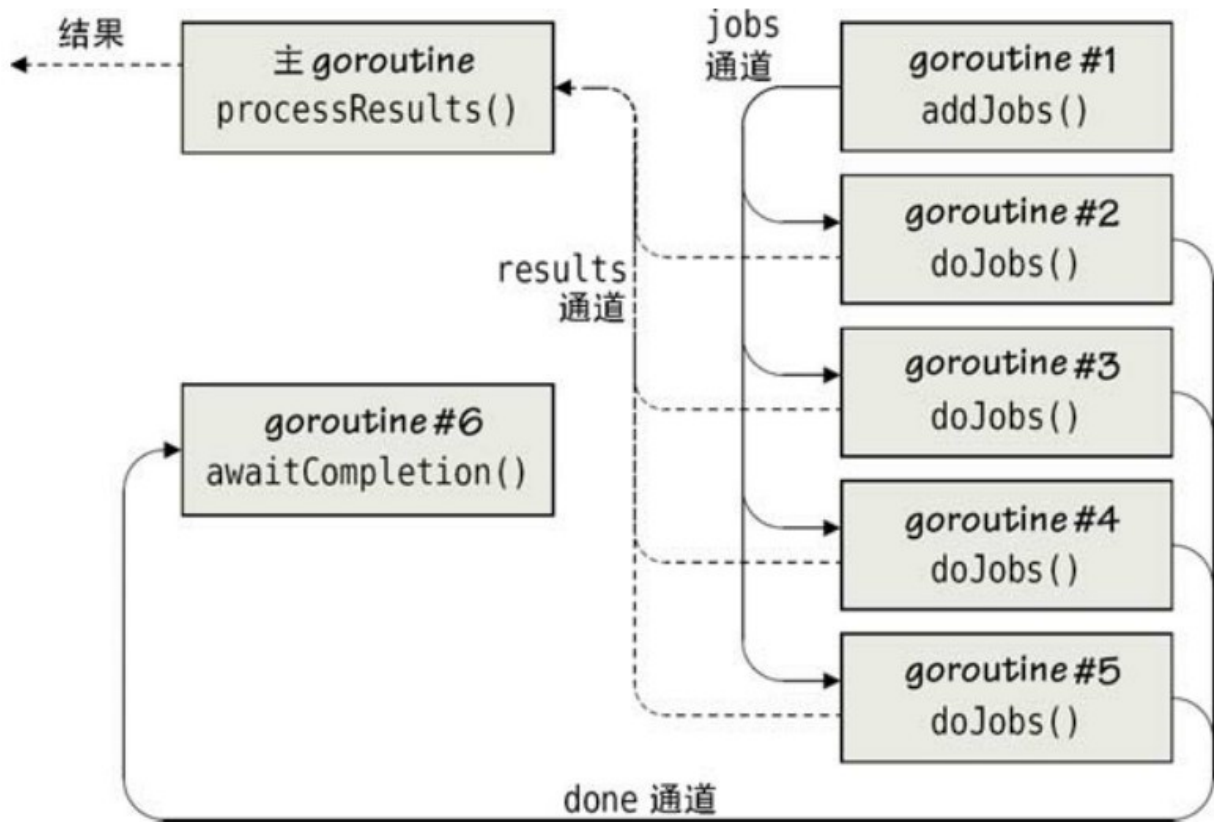


图7-5 多个独立的并发作业

有了通道之后，我们开始调用 `addJobs()` 函数往 `jobs` 通道里增加工作，这个函数也是在一个单独的 `goroutine` 里运行的。再调用 `doJobs()` 函数来执行实际的工作，实际上我们调用了这个函数四次，也就是创建了 4 个独立的 `goroutine`，各自做自己的事情。然后我们调用 `awaitCompletion()` 函数，它在自己的 `goroutine` 里等待所有的工作完成后关闭 `results` 通道。最后，我们调用 `processResults()` 函数，这个函数是在主 `goroutine` 里执行的，这个函数处理从 `results` 通道接收到的结果，当通道里没有结果时就会阻塞，直到接收完所有的结果才继续执行。图7-5展示了这个程序并发部分的语义。

```

func addJobs(jobs chan<- Job, filenames []string, results chan<- Result)
{
    for _, filename := range filenames {
        jobs <- Job{filename, results}
    }
    close(jobs)
}

```

这个函数将文件名一个接一个地以Job类型发送到jobs通道里。jobs通道有一个大小为4的缓冲区（和工作goroutine的数量一样），所以最开始那4个工作是立即就增加到通道里去的，然后该函数所在的goroutine阻塞等待其他的工作 goroutine 从通道里接收工作，以腾出通道空间来发送其他的工作。一旦所有的工作发送完毕（这取决于有多少个文件名需要处理，和处理每一个文件名的时间多长），jobs通道会被关闭。

尽管实际上传入的两个通道是双向的，但是我们将它们都指定为单向只允许发送的通道，因为我们在函数里就是这样使用的，Job结构里Job.results通道也是这么定义的。

```

func doJobs(done chan<- struct{}, lineRx *regexp.Regexp, jobs <-chan Job) {
    for job := range jobs {
        job.Do(lineRx)
    }
    done <- struct{}{}
}

```

前面我们已经知道，分别有4个独立的goroutine在执行doJobs()函数，它们都共享同一个jobs通道（只读），并且每个goroutine都会阻塞到直到有一个工作分配给它。拿到工作之后调用这个工作的Job.Do()方

法（很快我们就可以看到Do()方法里做了什么事情），当一个调用遍历完jobs之后，往done通道里发送一个空的结构报告自己的完成状态。

顺便提一下，按照 Go语言的惯例，带有通道参数的函数，通常会将目标通道放在前面，接下来才是源通道。

```
func awaitCompletion(done <-chan struct{}, results chan Result) {  
    for i := 0; i < workers; i++ {  
        <-done  
    }  
    close(results)  
}
```

这个函数（以及processResults()函数）确保主goroutine在所有的处理都完成后才退出，这样可以避免我们在前一节里提到过的陷阱（7.1节）。这个函数在它自己的goroutine里运行，然后等待从done通道里接收所有工作goroutine的完成状态，等待过程中它是阻塞的。一旦退出循环后 results 通道也会被关闭，因为这个函数能知道什么时候接收最后一个结果。注意这里我们不能将results通道作为一个只允许接收的通道（<-chan Result）来传给函数，因为Go语言不允许关闭这样的通道。

```
func processResults(results <-chan Result) {  
    for result := range results {  
        fmt.Printf("%s:%d:%s\n", result.filename, result.lino, result.line)  
    }  
}
```

这个函数是在主goroutine里执行的，遍历results通道或者阻塞在那里，一旦接收并处理完（例如打印）所有的完成状态后，循环结束，函数就会返回，然后整个程序将退出。

Go语言的并发支持是相当灵活的，这里我们使用的方法是，等待所有工作完成，关闭通道，并输出所有的结果，但我们还有其他的方

法。例如，`cgrep2`（它的文件在`cgrep2/cgrep.go`文件里）这个程序就是我们这一节讨论的 `cgrep1` 的另一个变种，它并没有使用 `awaitCompletion()` 或者 `precessResults()` 函数，只用了一个 `waitAndProcessResults()`函数。

```
func waitAndProcessResults(done <-chan struct{}, results <-chan
Result) {
    for working := workers; working > 0; {
        select { // 阻塞
            case result := <-results:
                fmt.Printf("%s:%d:%s\n",      result.filename,      result.lino,
result.line)
            case <-done:
                working--
        }
    }
    DONE:
    for {
        select { // 非阻塞
            case result := <-results:
                fmt.Printf("%s:%d:%s\n",      result.filename,      result.lino,
result.line)
            default:
                break DONE
        }
    }
}
```


这个函数首先就是一个for循环，它会一直执行到所有的goroutine退出。每一次进入循环体里都会执行select语句，然后阻塞直到接收到一个结果值或者一个完成值。（如果我们使用了一个非阻塞的select，也就是有一个default分支，相当于创建了一个非常省CPU的spin-lock。）当所有的goroutine退出后for循环也会结束，也就是说，所有的工作goroutine都往done通道里发送了一个结果值。

所有的工作goroutine完成自己的任务后，我们启动了另一个for循环，在这个循环里我们使用了非阻塞的select。如果results通道里还有未处理的值，select就会匹配第一个分支，将这个值输出，然后再一次执行循环体，一直重复到results通道里所有的值都被处理完毕。但如果这时没有结果值需要接收（最明显的就是刚进入for循环的时候results通道里是空的），程序就会退出for循环然后跳转到DONE标签处（单纯使用一个break语句是不够的，它只会跳出select语句），这个for循环不是很耗CPU，因为每一次迭代要么接收了一个结果值要么我们就完成了，没有不必要的等待时间。

实际上 waitAndProcessResults()函数要比原先的awaitCompletion()和process Results()函数更长和更复杂一点。但是，当有好几个不同的通道需要处理的时候，使用select语句是非常有好处的。例如我们可以在一定时间之后停止处理，即使那时还有未完成的任务。

下面是这个程序的第三个版本，也是最后一个版本，cgrep3（在文件cgrep3/cgrep.go里）。

```
func waitAndProcessResults(timeout int64, done <-chan struct{},
results <-chan Result) {
    finish := time.After(time.Duration(timeout))
    for working := workers; working > 0; {
        select { // 阻塞
        case result := <-results:
```

```

        fmt.Printf("%s:%d:%s\n",      result.filename,      result.lino,
result.line)
    case <-finish:
        fmt.Println("timed out")
        return // 超时， 因此直接返回
    case <-done:
        working--
    }
}
for {
    select { // 非阻塞
        case result := <-results:
            fmt.Printf("%s:%d:%s\n",      result.filename,      result.lino,
result.line)
            case <-finish:
                fmt.Println("timed out")
                return // 超时， 因此直接返回
            default:
                return
        }
    }
}

```

这是cgrep2的一个变种，不同的就是多了一个超时的参数，将一个time.Duration（其实就是一个纳秒值）值传入time.After()函数，返回一个超时通道。这个超时通道的作用就是超过了time.Duration指定的时间后，通道会返回一个值，如果我们从这个通道里读到一个值，也就是说超时了。这里我们将返回的通道赋值给finish变量，并在两个for循环

里为 `finish` 增加一个 `case` 分支。一旦超时（即 `finish` 通道发送了一个值），即使还有工作未完成，函数也会返回，然后程序结束。

如果在超时之前我们得到了所有的结果值，也就是所有的工作 `goroutine` 都完成自己的任务并向 `results` 通道发送了一个结果值，这时第一个 `for` 循环就会退出，程序接着执行第二个 `for` 循环，这过程和 `cgrep2` 是完全一样的，唯一不同的是这里并没有直接从 `for` 循环中跳出，而是简单地在默认分支里执行一个 `return` 语句，还增加了一个超时的 `case` 分支。

现在我们已经知道并发是怎么处理的了，下面的代码是关于每个工作是怎么被处理的，这个之后 `cgrep` 例子的所有的代码我们就已经讲解完了。

```
func (job Job) Do(lineRx *regexp.Regexp) {
    file, err := os.Open(job.filename)
    if err != nil {
        log.Printf("error: %s\n", err)
        return
    }
    defer file.Close()
    reader := bufio.NewReader(file)
    for lino := 1; ; lino++ {
        line, err := reader.ReadBytes('\n')
        line = bytes.TrimRight(line, "\n\r")
        if lineRx.Match(line) {
            job.results <- Result{job.filename, lino, string(line)}
        }
        if err != nil {
            if err != io.EOF {
```

```

        log.Printf("error:%d: %s\n", lino, err)
    }
    break
}
}
}

```

这个方法是用来处理每一个文件的，它传入一个`*regexp.Regexp`值，这是一个线程安全的指针，所以它不必关心有多少个不同的goroutine同时在使用它。整个函数的代码我们已经很熟悉了：打开一个文件，读取它的数据，对所有的出错进行处理，如果没有错误我们就用`defer`语句来关闭文件，然后创建了一个带缓冲区的`reader`来遍历文件内容里的所有行，一旦遇到了匹配的行，我们就将它作为一个`Result`值发送到`results`通道，当通道满时发送操作会被阻塞，最后所有被处理的文件都会产生`N`个结果值，如果文件里没有匹配的行，那`N`的值为0。

在Go语言里处理文本文件时，如果在读一行文本中出现错误，我们会在处理完当前行后处理这个错误。如果`bufio.Reader.ReadBytes()`方法遇到了一个错误（包括文件结束），它会和错误一起返回出错前已经成功读取到的字节数。有时候文件最后一行不是以换行符结束的，所以为了确保我们处理最后一行（不管它是否是以换行符结束的），我们都会在处理完这一行后再处理相关错误。这样做有一点不好，就是正则表达式如果匹配了一个空的字符串，我们会得到既不是`nil`也不是`io.EOF`的错误，从而被当做一个假的匹配（当然，我们有办法绕过这个问题）。

`bufio.Reader.ReadBytes()`方法会一直读到一个指定的字符后才返回。返回的字节流里包括那个指定的字符，如果整个文件都没出现这个字符的话，会将整个文件的数据都读取出来。我们这里不需要换行符，所以我们使用`bytes.TrimRight()`方法将它去掉。`bytes.TrimRight()`方

法的作用就是从行的右边向左去除指定的字符串或字符（类似于 `strings.TrimRight()` 函数）。为了能让我们的程序跨平台，我们将换行和回车字符都除掉。

另一个需要注意的小细节就是，我们读出来的是字节切片，而 `regexp.Regexp.Match()` 和 `regexp.Regexp.MatchString()` 方法只能处理字符串，所以我们将 `[]byte` 转换成 `string` 类型，当然转换的代价很小。还有我们统计行数从1开始而不是从0开始，这样会方便很多。

`cgrep` 程序的设计中比较好的一点就是它的并发框架足够简单，并和实际的业务处理过程（也就是 `Job.Do()` 方法）分离，只使用 `results` 通道来进行交互。这种框架与业务的分离在 Go 语言的并发编程里是很常见的，与那些使用底层同步数据结构（如同步锁）的方法相比有诸多好处，因为锁相关的代码会让程序的逻辑变得更加复杂和晦涩难懂。

7.2.3 线程安全的映射

Go 语言标准库里的 `sync` 和 `sync/atomic` 包提供了创建并发的算法和数据结构所需要的基础功能。我们也可以将一些现有的数据结构变成线程安全，例如映射或者切片等（参见 6.5.3 节），这样可以确保在使用上层 API 时所有的访问操作都是串行的。

这一节我们会开发一个线程安全的映射，它的键是字符串，值是 `interface{}` 类型，不需要使用锁就能够被任意多个 `goroutine` 共享（当然，如果我们存的值是一个指针或引用，我们还必须得保证所指向的值是只读的或对于它们的访问是串行的）。线程安全的映射的实现在 `safemap/safemap.go` 文件里，包含了一个导出的 `SafeMap` 接口，以及一个非导出的 `safeMap` 类型，`safeMap` 实现了 `SafeMap` 接口定义的所有方法。下一节我们来看看这个 `safeMap` 是怎么使用的。

安全映射的实现其实就是在在一个goroutine里执行一个内部的方法以操作一个普通的map数据结构。外界只能通过通道来操作这个内部映射，这样就能保证对这个映射的所有访问都是串行的。这种方法运行着一个无限循环，阻塞等待一个输入通道中的命令（即“增加这个”，“删除那个”等）。

我们先看看 SafeMap 接口的定义，再分析内部 safeMap 类型可导出的方法，然后就是safemap包的New()函数，最后分析未导出的safeMap.run()方法。

```
type SafeMap interface {
    Insert(string, interface{})
    Delete(string)
    Find(string) (interface{}, bool)
    Len() int
    Update(string, UpdateFunc)
    Close() map[string]interface{}
}

type UpdateFunc func(interface{}, bool) interface{}
```

这些都是SafeMap接口必须实现的方法。（我们在前一章讨论过可导出的接口和不能导出的具体类型是什么样的。）

UpdateFunc类型让自定义更新操作函数变得很方便，我们会在后面讨论Update()方法时讲到它。

```
type safeMap chan commandData
type commandData struct {
    action  commandAction
    key     string
    value   interface{}
    result  chan<- interface{}}
```

```

data      chan<- map[string]interface{}
updater UpdateFunc
}
type commandAction int
const (
    remove commandAction = iota
    end
    find
    insert
    length
    update
)

```

`safeMap`的实现基于一个可发送和接收`commandData`类型的通道。每个`commandData`类型值指明了一个需要执行的操作（在 `action` 字段）及相应的数据，例如，大多数方法需要一个`key`来指定需要处理的项。我们会在分析`safeMap`的方法时看到所有的字段是如何被使用的。

注意，`result`和`data`通道都是被定义为只写的，也就是说，`safeMap`可以往里面发送数据，不能接收。但是下面我们会看到，这些通道在创建的时候都是可读写的，所以它们能够接收`safeMap`发给它们的任何值。

```

func (sm safeMap) Insert(key string, value interface{}) {
    sm <- commandData{action: insert, key: key, value: value}
}

```

这种方法相当于一个线程安全版本的`m[key] = value`操作，其中 `m` 是 `map[string] interface{}` 类型。它创建了一个`commandData`值，指明是一个`insert`操作，并将传入的`key`和`value`保存到`commandData`结构中并发送到一个安全的映射里。我们刚刚介绍过，这个安全映射的类型是基

于chan commandData实现的（我们在6.4节讲过，在Go语言里创建一个结构时所有未被显式初始化的字段都会被默认初始化成它们各自的零值）。

当我们查看 safemap 包里的New()函数时我们会发现该函数返回的 safeMap 关联了一个goroutine。safeMap.run()方法在这个goroutine里执行，也是一个捕获了该safeMap通道的闭包。safeMap.run()里有一个底层 map 结构，用来保存这个安全映射的所有项，还有一个 for循环遍历 safeMap通道，并执行每一个从safeMap通道接收到的对底层map的操作。

```
func (sm safeMap) Delete(key string) {  
    sm <- commandData{action: remove, key: key}  
}
```

这个方法告知该安全映射删除key所对应的项，如果给定key不存在则不做任何事。

```
type findResult struct {  
    value interface{}  
    found bool  
}  
  
func (sm safeMap) Find(key string) (value interface{}, found bool) {  
    reply := make(chan interface{})  
    sm <- commandData{action: find, key: key, result: reply}  
    result := (<-reply).(findResult)  
    return result.value, result.found  
}
```

safeMap.Find() 方法创建了一个 reply 通道用来接收发送 commandData后的响应，然后把这个 reply 通道和指定要查找的key 放到一个 commandData 值里，再往 safeMap发送一个 find 命令。因为所

有的通道都没有带缓冲区，因此一条命令的发送操作会一直阻塞直到没有其他的goroutine往里面发送命令。一旦命令发送完毕我们立即接收reply通道的返回值（对应于find命令是一个findResult结构），然后将这个结果返回给调用者。顺便提一句，这里我们使用命名返回值是为了让它们的用途更加清晰。

```
func (sm safeMap) Len() int {
    reply := make(chan interface{})
    sm <- commandData{action: length, result: reply}
    return (<-reply).(int)
}
```

这个方法和Find()方法在结构上大体是相似的，首先创建一个用来接收结果的reply通道，最后将结果分析出来返回给调用者。

```
func (sm safeMap) Update(key string, updater UpdateFunc) {
    sm <- commandData{action: update, key: key, updater: updater}
}
```

这个方法貌似看起来有点不太常规，因为它的第二个参数是一个签名为func(interface{}, bool)的函数。Update方法往通道发送一条更新命令时会带上指定的key和一个updater函数，当这条命令被接收时，updater函数会被调用并带上两个调用参数，一个是指定的key对应的值（若key不存在，则传nil作为参数），还有一个bool变量表示这个键对应的项是否存在。指定的键对应的值会被设置为updater函数的返回值（如果key不存在则创建一个新项）。

需要特别注意的是，updater函数调用safeMap的方法会导致死锁。后面涉及safemap.safeMap.run()方法时会进一步解释。

但我们为什么需要这么奇怪的方法呢，又怎么去用它？

当我们需要插入、删除或查找safeMap里的项时，Insert()、Delete()和Find()方法都能工作得很好。但当我们想去更新一个已经存在的项时

会发生什么呢？举个例子，我们在`safeMap`里保存了机器零件的价格，现在我们需要将某个零件的价格上调5%，会发生什么事情呢？我们知道Go语言会自动将一个未初始化的值初始化为0，如果我们指定的键已经存在的话，它对应的值会增加5%，如果不存在的话，就创建一个新的零值。下面我们实现了一个能保存`float64`类型的值的安全映射。

```
if price, found := priceMap.Find(part); found { // 错误!
    priceMap.Insert(part, price.(float64)*1.05)
}
```

这段代码的问题是可能会有多个goroutine同时使用这个`priceMap`，也就有可能在`Find()`和`Insert()`之间修改数据，从而没法保证我们插入的价格值确实比原来的值高5%。

我们需要的是一个原子的更新操作，也就是说，读和更新这个值应该作为不可中断的一个操作。下面的`Update()`方法就是这样做的。

```
priceMap.Update(part, func(price interface{}) bool {
    if found {
        return price.(float64) * 1.05
    }
    return 0.0
})
```

这段代码实现了一个原子更新操作，如果指定的键不存在，我们就创建一个新的项，它的值为0.0，否则我们就将这个键对应的值增加5%。因为这个更新是在`safeMap`的goroutine里执行的，这期间不会有其他的命令被执行（例如，从其他goroutine发送过来的命令）。

```
func (sm safeMap) Close() map[string]interface{} {
    reply := make(chan map[string]interface{})
    sm <- commandData{action: end, data: reply}
    return <-reply
}
```

```
}
```

Close()方法的工作原理和Find()以及 Len()方法类似，不过它有两个不同的目标。首先，它需要关闭safeMap通道（在safeMap.run()方法里），这样就不会再有其他的更新操作。关闭safeMap通道将导致safeMap.run()方法里的for循环退出，进而释放相应于自动垃圾收集的goroutine。第二个目标是将底层的map[string]interface{}返回给调用者（如果调用者不需要，可以忽略它）。每一个safeMap只允许执行一次Close()方法，不管有多少个goroutine在访问，而且一旦Close()被调用就不能再调用任何其他方法。我们可以保留Close()方法返回的map并像使用一个普通map一样使用它，但只能在一个goroutine里使用。

到这里我们已经分析了safeMap所有导出的方法，最后一个我们要分析的是safemap包的New()函数，New()函数的作用就是创建一个safeMap并以SafeMap接口的方式返回，并执行safeMap.run()方法使用通道，提供一个map[string]interface{}用来保存实际的数据，并且处理所有的通信。

```
func New() SafeMap {  
    sm := make(safeMap) // safeMap 类型 chan commandData go  
    sm.run()  
    return sm  
}
```

safeMap实际上是chan commandData类型，所以我们必须使用内置的make()函数来创建一个通道并返回它的一个引用。有了safeMap之后我们调用它的run方法，在run()里还会创建一个底层映射用来保存实际的数据，run()在自己的goroutine里执行，执行go语句之后通常会立即返回。最后函数将这个safeMap返回。

```
func (sm safeMap) run() {  
    store := make(map[string]interface{})
```

```

for command := range sm {
    switch command.action {
    case insert:
        store[command.key] = command.value
    case remove:
        delete(store, command.key)
    case find:
        value, found := store[command.key]
        command.result <- findResult{value, found}
    case length:
        command.result <- len(store)
    case update:
        value, found := store[command.key]
        store[command.key] = command.updater(value, found)
    case end:
        close(sm)
        command.data <- store
    }
}
}

```

创建了一个用来存储的映射后，`run()`方法启动了一个无限循环来读取 `safeMap` 通道的命令，如果通道是空的，就一直阻塞在那里。

因为`store`是一个再普通不过的映射，所以接收到每一个命令该怎么处理就怎么处理，非常容易理解。另一个稍微不同的是更新操作。某个键所对应项的值将被设置成`command.updater()`函数的返回值。最后一个`end`分支对应`Close()`调用，首先关闭通道以防止再接收其他的命令，然后将存储映射返回给调用者。

前面我们提过如果 `command.updater()` 函数要是调用了 `safeMap` 的方法就会发生死锁，这是因为如果 `command.updater()` 函数不返回，`update` 这个分支就不能正常结束。如果 `updater()` 函数调用了一个 `safeMap()` 方法，它会一直阻塞到 `update` 分支完成，这样两个都完成不了。图7-2解释了这种死锁。

显然，使用一个线程安全的映射相比一个普通的 `map` 会有更大的内存开销，每一条命令我们都需要创建一个 `commandData` 结构，利用通道来达到多个 `goroutine` 串行化访问一个 `safeMap` 的目的。我们也可以使用一个普通的 `map` 配合 `sync.Mutex` 以及 `sync.RWMutex` 使用以达到线程安全的目的。另外还有一种方法就是如同相关理论所描述的那样创建一个线程安全的数据结构（例如，参见附录 C）。还有一种方法就是，每个 `goroutine` 都有自己的映射，这样就不需要同步了，然后在最后将所有 `goroutine` 的结果合并在一起即可。尽管方法很多，这里所实现的安全映射不但易用而且足以应对各种的场景。下一小节我们会看到如何应用这个 `safeMap`，并顺带与一些其他方法进行了对比。

[7.2.4 Apache报告](#)

并发处理最常见的一个需求就是更新共享数据。一个常见的方案是使用互斥量来串行化所有的数据访问。在Go语言里，我们除互斥量外还可以使用通道来达到串行化的目的。这一节，我们将使用通道和一个安全的映射（上一节讲过的）来开发一个小程序，然后再分析如何使用以互斥量保护的共享 `map` 来达成同样的目标。最后，我们将讲解如何使用通道局部的 `map` 来避免访问串行化从而最大化吞吐量，并使用通道来对一个 `map` 进行更新。

这里所有的工作都由 `apachereport` 程序完成。它读取从命令行指定的Apache网页服务器的 `access.log` 文件数据，然后统计所有记录里每个

HTML页面被访问的次数。这个日志文件很容易就增长到很大，所以我们用了一个 `goroutine` 来读取每一行日志（每行一条记录），以及另外3个`goroutine`一起处理这些行。每读到一个HTML页面被访问的记录，就将它更新到映射里去，如果这个HTML是第一次访问，则映射里对应的计数器为1，然后每再发现一条记录，计数器做加一处理。所以尽管有多个独立`goroutine`同时处理这些行记录，但是它们所有的更新都是在同一个映射里进行的。不同版本的程序采取不同的方法来更新映射。

7.2.4.1 用线程安全的共享映射同步

现在我们来回顾下 `apachereport1` 程序（在文件 `apachereport1/apachereport.go`里），使用前一节开发的`safeMap`，所用到的并发数据结构如图7-6所示。

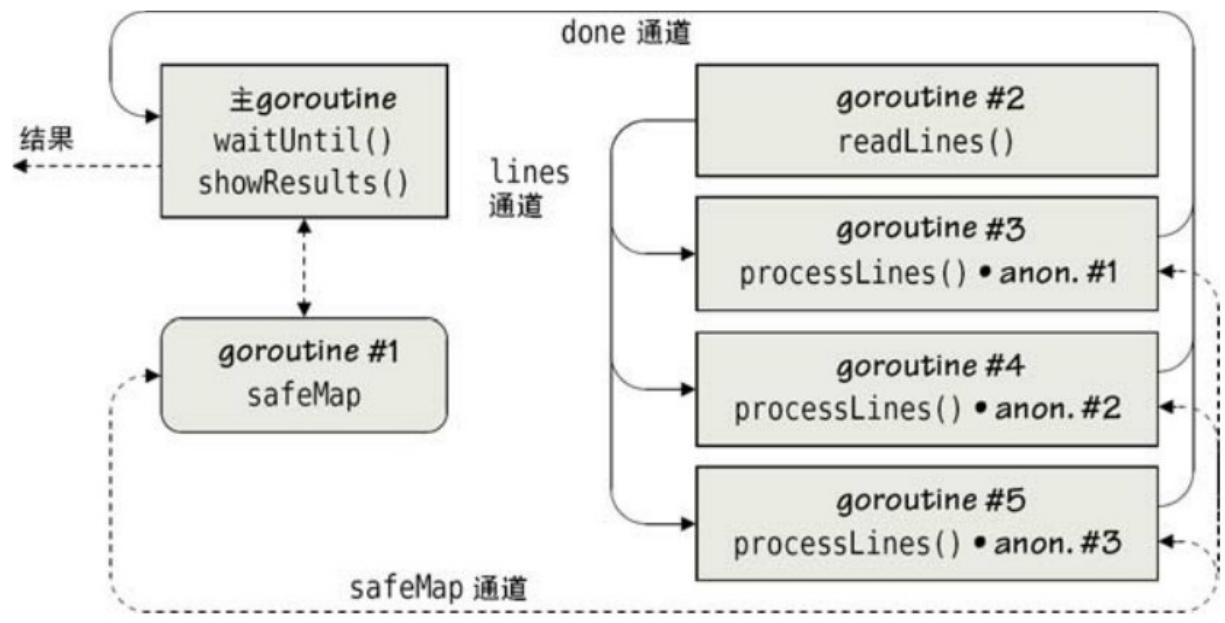


图7-6 带有同步结果的多个相互依赖的并发作业

在图7-6中， `goroutine#2`创建了一个通道， 将从日志读到的每一行发送到工作通道里， 然后`goroutine#3`到`goroutine#5`处理这个通道的每一行并更新到共享的`safeMap`数据结构。对`safeMap`的操作本身是在一个独立的`goroutine`里完成的， 所以整个程序一共使用了6个`goroutine`。

```

var workers = runtime.NumCPU()
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU()) // 使用所有的机器
内核
    if len(os.Args) != 2 || os.Args[1] == "-h" || os.Args[1] == "--help" {
        fmt.Printf("usage: %s <file.log>\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    lines := make(chan string, workers*4)
    done := make(chan struct{}, workers)
    pageMap := safemap.New()
    go readLines(os.Args[1], lines)
    processLines(done, pageMap, lines)
    waitUntil(done)
    showResults(pageMap)
}

```

`main()`函数首先确保Go运行时系统充分利用所有的处理器，然后创建两个通道来组织所有的工作。从日志文件里读取到的每一行将被发送到`lines`通道，然后工作goroutine再将每一行读取出来进行处理，我们为`lines`通道分配了一个小缓冲区以降低工作goroutine阻塞在`lines`通道上的可能性。`done`通道用来跟踪何时所有工作被完成。因为我们只关心发送和接收操作的发生而非实际传递的值，所以我们使用一个空结构。`done`通道也是带有缓冲的，所以当有一个goroutine报告工作完成时不会被阻塞在发送操作上。

接着我们使用`safemap.New()`函数创建了一个`pageMap`，它是一个非导出的`safeMap`类型的值，实现了`SafeMap`接口所有定义的方法，可以随意传递。然后我们启动一个goroutine来从日志文件里读取行记录，并

启动其他的goroutine 负责处理这些行。最后程序等待所有的goroutine 工作完成，并将最终的结果输出。

```
func readLines(filename string, lines chan<- string) {
    file, err := os.Open(filename)
    if err != nil {
        log.Fatal("failed to open the file:", err)
    }
    defer file.Close()
    reader := bufio.NewReader(file)
    for {
        line, err := reader.ReadString('\n')
        if line != "" {
            lines <- line
        }
        if err != nil {
            if err != io.EOF {
                log.Println("failed to finish reading the file:", err)
            }
            break
        }
    }
    close(lines)
}
```

这个函数看起来并不陌生，因为与之前我们见过的几个例子相当类似。首先最关键的第一个地方就是我们将每一个文本行发送到 **lines** 通道，**lines** 是只允许发送的，而且当通道缓冲区满了之后这个操作会一直阻塞在那里，直到有一个其他的goroutine从通道里接收一个文本

行。不过就算有阻塞，也只会对这个 `goroutine` 有影响，其他的 `goroutine` 还会继续工作而不受影响。第二个关键的地方就是当所有的文本行发送完毕之后我们关闭`lines`通道，这就告诉了其他的`goroutine`已经没有数据需要接收了。记住，尽管这个`goroutine`和其他的`goroutine`（也就是其他负责处理任务的工作 `goroutine`）是并发执行的，但是一般只有在大部分工作完成后`close()`语句才会被执行到。

```
func processLines(done chan<- struct{}, pageMap safemap.SafeMap,
lines <-chan string) {
    getRx := regexp.MustCompile('GET[ \\t]+([^\t\n]+[.]html?)')
    incremter := func(value interface{}, found bool) interface{} {
        if found {
            return value.(int) + 1
        }
        return 1
    }
    for i := 0; i < workers; i++ {
        go func() {
            for line := range lines {
                if matches := getRx.FindStringSubmatch(line); matches != nil
                {
                    pageMap.Update(matches[1], incremter)
                }
            }
            done <- struct{}{}
        }()
    }
}
```

这里函数参数的顺序遵循Go语言的约定，先是目标通道（也就是done通道），然后是源通道（lines通道）。

该函数创建了一些goroutine（实际上是3个）来处理实际的工作。每个goroutine都共享同一个*regexp.Regexp数据（和普通的指针不同，这个是线程安全的）和一个incrementer()函数（这个函数不会有任何副作用，因为它不访问任何共享的数据），还共享了同一个pageMap（是一个SafeMap接口类型的值）。前面我们已经知道，safeMap的修改都是线程安全的。

如果没有匹配任何数据，那么 regexp.Regexp.FindStringSubmatch() 函数返回nil，否则就返回一个[]string 类型的字符串切片，其中第一个字符串是整个正则表达式的匹配，随后其他的字符串对应表达式里的每一个小括号括起来的子表达式。这里我们只有一个子表达式，所以如果我们得到一个匹配的结构，那这个结果里有两个字符串，一个是完整的匹配，另一个是括号里子表达式的匹配，在这里是HTML页面的文件名。

每一个工作 goroutine 从只允许接收的lines 通道里读取文本行，通道中的数据由readLines()函数里的goroutine 从日志文件里读取并发送。对于某一行的匹配说明对于HTML文件发生了一个GET请求，在这种情况下 safeMap.Update()方法将被调用并传入页面的文件名（也就是matches[1]）和incrementer()函数。incrementer()函数是safeMap的内部goroutine 调用的，对于之前被访问过的页面，那就返回一个增量值，对于未被访问过的页面则返回1（回忆起前一小节我们说过的，如果被传给safeMap.Update()的函数自身又调用了 safeMap的其他方法的话会出现死锁）。当所有页面被处理后，每一个工作goroutine会发送一个空结构体到done通道以说明工作已经完成。

```
func waitUntil(done <-chan struct{}) {  
    for i := 0; i < workers; i++ {
```

```

        <-done
    }
}

```

这个函数在主 `goroutine` 里执行，阻塞在 `done` 通道上，当所有的工作 `goroutine` 往 `done`里发送了一个空结构体后，`for`循环将结束。和平时一样，我们不需要关闭`done`通道，因为没有在别的需要检查这个通道是否被关闭的地方使用这个通道。通过阻塞，这个函数可以确保所有的处理工作在主`goroutine`退出之前完成。

```

func showResults(pageMap safemap.SafeMap) {
    pages := pageMap.Close()
    for page, count := range pages {
        fmt.Printf("%8d %s\n", count, page)
    }
}

```

当所有的文本行都被读取并且所有的匹配项都增加到 `safeMap` 之后，该函数将被调用以输出结果。它首先调用 `safemap.safeMap.Close()` 方法关闭 `safeMap`的通道，退出在`goroutine`里运行的`safeMap.run()`方法，然后返回一个底层的`map[string]interface{}`给调用者。这个返回的映射将无法再被其他的`goroutine`通过安全映射的通道访问，所以可以在一个单独的`goroutine`中安全地使用它（或者使用互斥量来让多个`goroutine`串行访问）。由于从该处之后我们只在主`goroutine`里访问这个映射，所以串行化访问并没必要。我们简单地遍历映射里所有的“键/值”对，然后将它们输出到控制台。

使用一个 `SafeMap` 接口类型的值同时提供了线程安全性和简单的语法，不需要担心锁的问题。这种方法不好的一点就是安全映射的值是 `interface{}` 类型而不是一个特定的类型，这样我们就得在

incrementer()函数里使用类型断言（我们将在7.2.4.3节讨论另一个缺陷）。

7.2.4.2 用带互斥量保护的映射同步

现在我们将对简单干净的基于通道的做法和传统的基于互斥量的做法做一个对比。为此我们首先简要地讨论一下apachereport2程序（在文件apachereport2/apachereport.go里）。这个程序是 apachereport1的变种，使用了一个封装了映射的自定义数据类型和互斥量来取代线程安全的映射。这两个程序所做的工作是完全一样的，唯一不同的是映射的值是一个int型值而不是SafeMap里的interface{}类型，并且相比安全映射中的完全方法列表,这里只提供了这个工作相关的最小功能集合——一个Increment()方法。

```
type pageMap struct {  
    countForPage map[string]int  
    mutex         *sync.RWMutex  
}
```

使用自定义类型的好处就是我们可以用所需要的特定数据类型而不是通用的interface{}类型。

```
func NewPageMap() *pageMap {  
    return &pageMap{make(map[string]int), new(sync.RWMutex)}  
}
```

这个函数返回一个可立即使用的*pageMap值。（顺便提一句，可以使用 &sync.RWMutex{} 来创建一个读写锁，而不用 new(sync.RWMutex)。4.1节中我们讨论过这两者的一致性。）

```
func (pm *pageMap) Increment(page string) {  
    pm.mutex.Lock()  
    defer pm.mutex.Unlock()  
    pm.countForPage[page]++
```

```
}
```

每个修改 `countForPage` 的方法都需要使用互斥量来串行化访问。我们这里用的方法很传统：首先锁定互斥量，然后使用 `defer` 关键字来调用解锁互斥量的语句，这样无论什么时候返回都能保证可以解锁互斥量（即使发生了异常），然后再访问映射里的数据（每次锁定的时间越少越好）。

基于Go语言的自动初始化机制，当页面在 `countForPage` 中第一次被访问时（即该页面还不在 `countForPage` 里），我们就将它增加到这个映射里面并将值设置为0，然后马上递增该值。相对的，之后对已经存在于映射中的页面的访问都会导致对应值的递增。

我们使用互斥量来串行化所有方法对 `countForPage` 的访问，所以如果要更新映射的值，就必须使用 `sync.RWMutex.Lock()` 和 `sync.RWMutex.Unlock()`，但对于只读的访问，我们可以用另一种只读的方法。

```
func (pm *pageMap) Len() int {  
    pm.mutex.RLock()  
    defer pm.mutex.RUnlock()  
    return len(pm.countForPage)  
}
```

我们将这个放进来纯粹是为了展示一下如何使用一个读锁。这个用法和普通的锁是一样的，但读锁可能更加高效一点（因为我们承诺只是读取但不修改受保护的资源）。例如，如果我们有多个 `goroutine` 都同时读同一个的 `countForPage`，利用读锁，它们可以安全地并发执行。但如果它们其中一个得到了一个读写锁，它将可以修改映射的数据，但其余的 `goroutine` 就无法再获取任何锁。

```
pageMap.Increment(matches[1])
```

在有了pageMap类型后，工作goroutine就可以用这个语句来更新共享映射。

7.2.4.3 同步：使用通道来合并局部映射

不管我们用的是安全的映射还是用互斥量保护的映射，通过增加工作goroutine的数量可能能够提升应用程序的运行速度。但是由于访问安全映射或者用互斥量保护的映射时必须是串行化的，增加goroutine的数量会直接导致竞争的增加。

对于这种情况，通常我们可以通过牺牲一些内存来提升速度。例如，我们可以让每个工作goroutine都拥有自己的映射，这样可以极大地提高应用程序的吞吐量，因为处理过程中不会发生任何竞争，代价就是使用了更多的内存(因为很可能每个映射都有部分甚至所有相同的页面)。最后我们当然还必须将这些映射合并起来，这会是一个性能瓶颈，因为一个映射在合并时所有其他准备好合并的映射只能等着。

程序apachereport3（在文件apache3/apachereport.go里）使用每个goroutine特定的本地映射结构，并最后将它们全部合并到同一个映射里去。该程序的代码和apachereport1以及 apachereport2 几乎是一样的，这样我们就只重点介绍这个方法不一样的地方。这个程序的并发结构如图7-7所示。

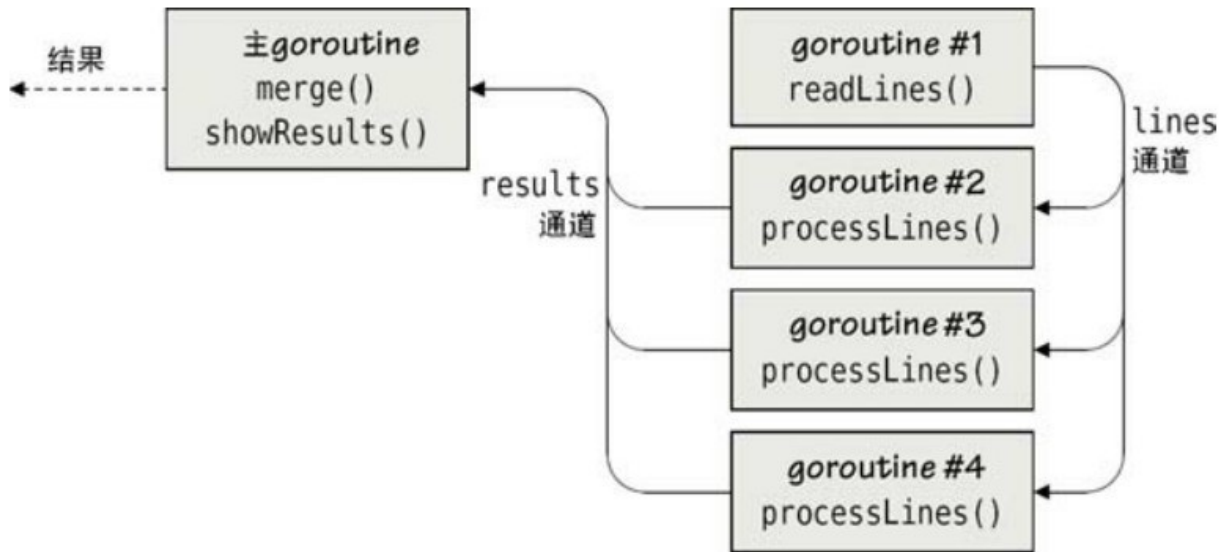


图7-7 带有同步结果的多个相互依赖的并发作业

```
//...
lines := make(chan string, workers*4)
results := make(chan map[string]int, workers)
go readLines(os.Args[1], lines)
getRx := regexp.MustCompile('GET[ \t]+([^\t\n]+[.].html?)')
for i := 0; i < workers; i++ {
    go processLines(results, getRx, lines)
}
totalForPage := make(map[string]int)
merge(results, totalForPage)
showResults(totalForPage)
//...
```

这是apachereport3程序main()函数的一部分。这里我们没有使用done通道而是使用了一个results通道，当每个goroutine处理完成之后，将本地生成的映射发送到results这个通道里。另外，我们还创建一个保存所有结果的映射（叫totalForPage）以保存所有合并所有的结果。

```

func processLines(results chan<- map[string]int, getRx
*regexp.Regexp,
lines <-chan string) {
    countForPage := make(map[string]int)
    for line := range lines {
        if matches := getRx.FindStringSubmatch(line); matches != nil {
            countForPage[matches[1]]++
        }
    }
    results <- countForPage
}

```

这个函数和前一个版本几乎是一样的，关键的区别有两个，第一个就是我们创建了一个本地映射来保存页面的数量，第二个就是在函数处理完所有的文本行之后（也就是`lines`通道被关闭了），我们将本地的映射结果发送到`results`通道（而不是发送一个`struct{}{}到done通道`）。

```

func merge(results <-chan map[string]int, totalForPage map[string]int)
{
    for i := 0; i < workers; i++ {
        countForPage := <-results
        for page, count := range countForPage {
            totalForPage[page] += count
        }
    }
}

```

`merge()`函数的结构和之前我们看过的`waitUntil()`是一样的，只是这一次我们需要使用接收到的值，用以更新 `totalForPage` 映射。需要注意

的是，这里接收的映射不会再被发送的goroutine访问，所以无需使用锁。

showResults()函数也基本上和之前的是一样的（所以这里就不贴代码了），我们将totalForPage作为它的参数，然后在函数里遍历这个映射，将每个页面的统计结果打印出来。

apachereport3程序的代码相对apachereport1和apachereport2来说非常的简洁，而且它所用的并发模型在很多场合是非常有用的，也就是每个goroutine都有局部的数据结构来保存计算结果，并将最后所有goroutine运行的结果合并在一块。

当然，对于那些习惯使用锁的程序员来说，大多还是倾向于使用互斥量来串行化共享数据的访问。但是，Go语言文档强烈推荐使用goroutine和通道，它提倡“不要使用共享内存来通信，相反，应使用通信来共享内存”，而且Go编译器对于上面提到的并发模型进行了相应的优化。

7.2.5 查找副本

这是这章最后一个关于并发的例子，使用SHA-1值而不是根据文件名来查找重复的文件 [2]。

我们即将分析的程序名字是 findduplicates（在文件 fundduplicates/findduplicates.go 里）。程序使用了标准库里的 filepath.Walk()函数，遍历一个给定路径的所有文件和目录，包括子目录、子目录的子目录等。程序根据工作量的多少而决定使用多少个goroutine。对于每一个大文件会有一个goroutine被单独创建以用于计算文件的SHA-1值，而小文件则是直接在当前的goroutine里计算。这意味着我们不知道实际会有多少个goroutine在运行，不过我们也可以设置一个上限。

怎么处理若干个不固定数量的goroutine呢，一种办法就是和之前的例子一样使用done通道，只不过这一次是用来监控所有goroutine的状态。使用sync.WaitGroup虽然容易，但是我们需要将goroutine的数量传给它，而goroutine的数量我们是不知道的。

```
const maxGoroutines = 100

func main() {
    runtime.GOMAXPROCS(runtime.NumCPU()) // 使用所有的机器
    核心
    if len(os.Args) == 1 || os.Args[1] == "-h" || os.Args[1] == "--help" {
        fmt.Printf("usage: %s <path>\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    infoChan := make(chan fileInfo, maxGoroutines*2)
    go findDuplicates(infoChan, os.Args[1])
    pathData := mergeResults(infoChan)
    outputResults(pathData)
}
```

main()函数从命令行读取一个路径作为处理起始点并安排所有之后的工作。它首先创建一个通道用来传送 fileInfo 值（我们很快就会看到）。我们为此通道设置了缓冲，因为实验表明这将能稳定的提升性能。

接下来函数在一个goroutine里执行findDuplicates()函数，并调用mergeResults()函数以读取infoChan通道里的数据直到它关闭。当合并结果返回后，我们将结果打印出来。

程序所有的goroutine和通信流程图如图 7-8 所示。图中的结果通道中的值是 fileInfo 类型的，这些值会被一个叫“walker”的函数（filepath.WalkFunc()类型）发送到infoChan通道，walker函数是我们调

用 `filepath.Walk()` 时传入的参数。`filepath.Walk()` 函数也是在 `fileDuplicates()` 里被调用的。`mergeResults()` 函数负责接收最后的结果。图中所示的 `goroutine` 是在 `findDuplicates()` 函数和 `walker` 函数里创建的。另外，标准库里的 `filepath.Walk()` 函数也会创建 `goroutine`（例如，每一个 `goroutine` 处理一个目录），至于它是怎么工作的则属于实现细节。

```
type fileInfo struct {  
    sha1 []byte  
    size int64  
    path string  
}
```

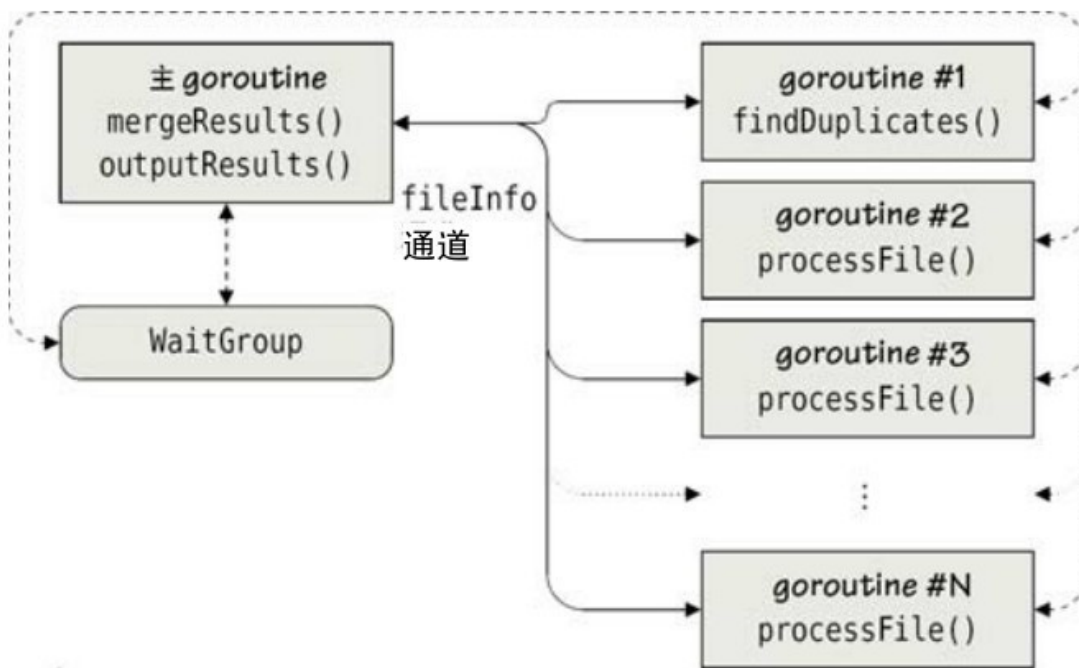


图7-8 带有同步结果的多个独立的并发作业

我们用这个结构体来保存文件的一些信息，如果两个文件的SHA-1值和文件尺寸都是一样的，不管它们的路径或者文件名是什么，我们都会把它们认为是重复的。

```
func findDuplicates(infoChan chan fileInfo, dirname string) {
```

```

waiter := &sync.WaitGroup{}
filepath.Walk(dirname, makeWalkFunc(infoChan, waiter))
waiter.Wait() // 一直阻塞到工作完成
close(infoChan)
}

```

这个函数调用`filepath.Walk()`来遍历一个目录树，并对于每一个文件或者目录调用作为该函数第二个参数传入的`filepath.Walk()`函数来处理。

`walker`函数会创建任意个goroutine，我们必须等所有的goroutine完成任务之后才可以返回`findDuplicates()`函数。为此，我们创建了一个`sync.WaitGroup`，每次我们创建一个goroutine时，就调用一次`sync.WaitGroup.Add()`函数，而当goroutine完成任务之后，再调用`sync.WaitGroup.Done()`。所有的goroutine都设置为正在运行后，我们调用`sync.WaitGroup.Wait()`函数来等待所有工作goroutine完成。`sync.WaitGroup.Wait()`将阻塞到宣布完成的done数量和添加的数量相等为止。

所有的工作goroutine都退出后将不会再有其他的fileInfo值发送到infoChan里，因此我们可以关闭infoChan通道。当然`mergeResults()`仍然可以读这个通道，直到将所有的数据都被读取出来。

```

const maxSizeOfSmallFile = 1024 * 32
func makeWalkFunc(infoChan chan fileInfo, waiter *sync.WaitGroup)
func(string, os.FileInfo, error) error {
    return func(path string, info os.FileInfo, err error) error {
        if err == nil && info.Size() > 0 && (info.Mode()&os.ModeType
        == 0) {
            if info.Size() < maxSizeOfSmallFile || runtime.NumGoroutine()
            > maxGoroutines {

```

```

        processFile(path, info, infoChan, nil)
    } else {
        waiter.Add(1)
        go processFile(path, info, infoChan, func() { waiter.Done() })
    }
    return nil // 忽略所有错误
}
}
}

```

`makeWalkFunc()`创建了一个类型为 `filepath.WalkFunc`的匿名函数，原型为`func(string, os.FileInfo, error) error`。每当`filepath.Walk()`得到一个文件或者目录之后就会相应地调用这个匿名函数。函数中的`path`是指目录或者文件的名称，`info`保存了部分`stat`调用的结果，`err`要么为`nil`要么包含了详细的关于路径的错误信息。如果我们需要忽略目录，可以使用 `filepath.SkipDir` 作为 `error`的返回值，还可以返回其他`non-nil`的错误，这样`filepath.Walk()`函数就会终止返回。

这里我们只处理那些非零大小的正常文件（当然，所有文件大小为0都是一样的，不过我们忽略掉这些）。`os.ModeType`是一个位集合，包含了目录、符号连接、命名管道、套接字和设备，所以如果这些对应的位没有设置，那它就是一个普通的文件。

如果文件很小，如不到32 KB，我们使用自定义函数`processFile()`来计算它的SHA-1值，其他文件则创建一个新的`goroutine`来异步调用`processFile()`函数，这就意味着小的文件会被阻塞（直到我们计算出它们的SHA-1值），但大文件就不会，因为它们的计算是在一个独立的`goroutine`里完成的。总之，当所有的计算都完成了，作为结果的`fileInfo`值就会被发送到`infoChan`通道。

当我们创建一个新的 `goroutine` 之后，我们只需要调用 `sync.WaitGroup.Add()` 方法，但这么做的话，当 `goroutine` 完成自己的工作后还必须调用对应的 `sync.WaitGroup.Done()` 方法。我们利用 Go 语言的闭包来实现这个功能。如果我们在一个新的 `goroutine` 里调用 `processFile()` 函数，我们将一个匿名函数作为最后一个参数传入，当匿名函数被调用时会调用 `sync.WaitGroup.Done()` 方法，`processFile()` 函数应以延迟方式调用这个匿名函数，以保证当 `goroutine` 完成时 `Done()` 方法会被调用。如果我们在当前的 `goroutine` 里调用 `processFile()` 函数，我们传一个 `nil` 参数来代替匿名函数。

为什么我们不简单地为每一个文件都创建一个新的 `goroutine` 呢？在 Go 语言里完全可以这么做，就算我们创建了成百上千个 `goroutine` 也不会遇到任何问题。不幸的是，大部分的操作系统都限制同时打开的文件数。在 Windows 系统上默认只有 512，尽管能提升到 2048。Mac OS X 系统更低，只能同时打开 256 个文件，Linux 系统默认限制在 1024，但是这些类 Unix 操作系统通常可以将这个值设置成一万、十万或者更高。很明显，如果我们将每一个文件都放到单独的 `goroutine` 里去处理，就很容易会超出这个限制。

为了避免打开过多的文件，我们配合使用两个策略。首先，我们将所有的小文件都放在同一个 `goroutine` 里处理（或者几个 `goroutine`，如果碰巧 `filepath.Walk()` 将它的工作分散到几个 `goroutine` 里去处理然后并发地调用 `walker` 函数的话），这样就可以确保如果我们遇到了一个包含上千个小文件的目录，不需要一次打开太多的文件，因为一个 `goroutine` 或者几个就能很快地把它处理完。

我们还应该让大文件在单独一个 `goroutine` 里处理，因为大文件通常处理起来很慢，我们也就没有办法同时打开太多的大文件。所以我们的第二个策略就是，当有足够多的 `goroutine` 在运行后，我们就不再为处理大的文件创建新的 `goroutine` 了（`runtime.NumGoroutine()` 函数能告诉

我们在该函数调用的瞬间有多少goroutine在运行），而是强制让当前的goroutine直接处理后续的每一个文件，不管它的大小是多少，并同时监控当前正在运行的goroutine的总数，这也就相当于限制了我们同时打开的文件数。一个goroutine处理完大文件并被Go运行时系统移除后，goroutine的总数就会减少。这会导致有时goroutine总数低于我们限制的最大数量，这时我们可以再创建新的goroutine去处理大文件。

```
func processFile(filename string, info os.FileInfo, infoChan chan
fileInfo, done func()) {
    if done != nil {
        defer done()
    }
    file, err := os.Open(filename)
    if err != nil {
        log.Println("error:", err)
        return
    }
    defer file.Close()
    hash := sha1.New()
    if size, err := io.Copy(hash, file);
        size != info.Size() || err != nil {
        if err != nil {
            log.Println("error:", err)
        } else {
            log.Println("error: failed to read the whole file:", filename)
        }
        return
    }
}
```

```
    infoChan <- fileInfo{hash.Sum(nil), info.Size(), filename}
}
```

这个函数是由当前的goroutine或者一个新创建的goroutine调用的，用来计算给定的文件的SHA-1值，并将文件的详细信息发送到infoChan通道。

如果done变量不为nil，也就是说这个函数是在一个新创建的goroutine里调用的，我们要用defer来执行done()函数（它只是简单地调用一个sync.WaitGroup.Done()方法）。这也可以确保对于每一个sync.WaitGroup.Add()调用，都有一个对应的Done()调用，这对正确调用sync.WaitGroup.Wait()函数来说是必不可少的。如果done为nil，则我们忽略它。

接着，我们打开给定的文件进行读取，然后和往常一样用defer语句来关闭它。标准库里的crypto/sha1包提供了sha1.New()函数，sha1.New()返回一个实现了hash.Hash接口的值。这个接口提供了一个Sum()方法能得到一个hash值(也就是20字节的SHA-1值)，并且还实现了io.Writer接口定义的所有方法。（我们传进去一个nil让Sum()方法返回一个新的[]byte值，另外，我们也可以传一个已经存在的[]byte值进去，从而对hash值进行累加。）

我们可以先读取整个文件的内容，然后调用sha1.Write()方法将这个文件的数据传进去，作为计算散列值的一部分，但我们这里选择了一种更加高效的方法就是用io.Copy()函数，这个函数有两个参数：一个用于写数据的writer（这里是hash变量）和一个用于读取数据的reader（这里是打开的文件），并将reader中的数据复制到writer里。当复制完成时，io.Copy()返回成功复制的字节数，还有一个error类型的值，如果复制过程中没有出现任何问题，则这个值为nil，否则不为nil。因为SHA-1值是可以每次只计算一块数据的，所以io.Copy()缓冲区所用的最大内存就是SHA-1值的大小和一些固定的内存开销，如果我们将整个文

件读进内存，除了这些开销，还得有足够的内存来保存整个文件，所以，特别是对大文件，使用`io.Copy()`是非常节省的。

一旦我们计算出了SHA-1值，我们将`fileInfo`值发送到`infoChan`通道，`fileInfo`里保存了文件的SHA-1值、文件的大小（可以从`walker`里调用`processFile()`函数时传入的`os.FileInfo`值里得到）和文件名（包括完整的路径）。

```
type pathsInfo struct {  
    size int64  
    paths []string  
}
```

这个结构是用来保存每一个重复文件的详细信息的，也就是大小、所有文件的路径和文件名。主要在`mergeResults()`函数和`outputResults()`函数中用到。

```
func mergeResults(infoChan <-chan fileInfo) map[string]*pathsInfo {  
    pathData := make(map[string]*pathsInfo)  
    format := fmt.Sprintf("%%016X:%%dX", sha1.Size*2) // ==  
    "%016X:%40X"  
    for info := range infoChan {  
        key := fmt.Sprintf(format, info.size, info.sha1)  
        value, found := pathData[key]  
        if !found {  
            value = &pathsInfo{size: info.size}  
            pathData[key] = value  
        }  
        value.paths = append(value.paths, info.path)  
    }  
    return pathData
```

```
}
```

这个函数首先创建一个映射来保存重复的文件。该映射的键是字符串类型，由文件的大小、一个冒号和文件的SHA-1值组成，值就是*pathInfo信息。

为了构造相应的键，我们使用固定16个十六进制并用0作为对齐填充的数字来表达文件的大小，另外，还有足够的十六进制数字长度来表示文件的SHA-1值。因为键字符串的文件大小部分是在数字左边补0，所以之后我们还可以对键按照文件大小进行排序。sha1.Size常量表示SHA-1值占用的字节数，例如20个字节。因为一个字节在十六进制里需要用两个数字表示，所以格式化字符串的时候我们得准备一个两倍于SHA-1字符串字符数的存储空间。（我们也可以将格式化字符串写成这样：format = "%016X:%" + fmt.Sprintf("%dX", sha1.Size*2)。）

尽管有多个goroutine往infoChan通道里发送数据，这个函数是唯一从通道里读取数据的（它的主goroutine里执行），for循环接收fileInfo的值，或阻塞等待在那里。所有的值都接收完毕并且infoChan通道被关闭后，循环结束。对于每一个接收到的fileInfo值，计算它的键字符串，然后到映射里找这个键对应的项，如果找不到，就先用文件的大小和一个保存路径的空字符串切片来创建一个值，增加到这个新键对应的项里。然后，再将fileInfo值里文件的路径追加到这个项的值里去。

最后，如果文件是重复的，肯定会有多个路径，而不重复的文件只会有一个路径。一旦所有的fileInfo值处理完毕，函数将得到的映射返回以便后续继续处理。

```
func outputResults(pathData map[string]*pathsInfo) {  
    keys := make([]string, 0, len(pathData))  
    for key := range pathData {  
        keys = append(keys, key)
```

```

    }
    sort.Strings(keys)
    for _, key := range keys {
        value := pathData[key]
        if len(value.paths) > 1 {
            fmt.Printf("%d duplicate files (%s bytes):\n",
                len(value.paths), commas(value.size))
            sort.Strings(value.paths)
            for _, name := range value.paths {
                fmt.Printf("\t%s\n", name)
            }
        }
    }
}

```

`pathData` 映射的键字符串的头部用了 16 个十六进制数字的长度来保存文件的大小，并且使用 0 来填充（为什么使用十六进制呢，因为这样可以足够表示 `int64` 类型的值）。这就意味着通过对键进行排序，我们可以得到一个按照文件大小从最小到最大排序的文件列表。所以，这个函数首先创建了一个 `keys` 切片，将 `pathData` 里的所有键保存在这里，然后对 `keys` 排序，再遍历这个 `keys`，这样就可以得到 `pathsInfo` 里对应的 `value` 值。如果某个文件有多个路径，则我们还会对值进行排序再输出，如下所示。

```

$.findduplicates $GOROOT
2 duplicate files (67 bytes):
    /home/mark/opt/go/test/fixedbugs/bug248.dir/bug0.go
    /home/mark/opt/go/test/fixedbugs/bug248.dir/bug1.go
...

```

4 duplicate files (785 bytes):

/home/mark/opt/go/doc/gopher/gophercolor16x16.png

/home/mark/opt/go/favicon.ico

/home/mark/opt/go/misc/dashboard/godashboard/static/favicon.ico

/home/mark/opt/go/src/pkg/archive/zip/testdata/gophercolor16x16.png

...

2 duplicate files (1,371,249 bytes):

/home/mark/opt/go/bin/ebnflint

/home/mark/opt/go/src/cmd/ebnflint/ebnflint

上面的结果中我们省略了大部分的行。

```
func commas(x int64) string {  
    value := fmt.Sprint(x)  
    for i := len(value) - 3; i > 0; i -= 3 {  
        value = value[:i] + "," + value[i:]  
    }  
    return value  
}
```

通常对于太大的数字，我们很难一眼就能看出是多少，例如1 371 249，所以我们简单地使用`commas()`函数把数字用逗号分隔开，函数的参数必须是`int64`类型的，所以如果我们用的是`int`型或者其他大小的整数，我们得强制类型转换，例如`commas(int64(i))` [\[3\]](#)。

现在我们已经将整个`findduplicates`程序的代码和Go语言常见的并发编程都看过了。Go语言对并发的支持是非常灵活的，例如`<-`、`chan`、`go`、`select`等，而且还有很多其他的方法，不仅仅是我们书上提到的那些。尽管如此，这个例子，还有后面的练习，提供了大量帮助

大家理解Go语言并发编程的实例，大家可以自己动手写一个新的并发程序。

当然，我们没法衡量到底哪种方法才是最好用的，因为每种方法都有自身特定的应用场景。性能测试也会依赖于特定的机器、goroutine的数量，以及是否是纯粹的内存处理还是包括外部数据，如网络或者磁盘等。最可靠的方法就是用真实的数据来对不同的方法和不同数量的goroutine来进行测量，看它们的时间或者性能等。

7.3 练习

本章有3个练习，第一个是创建一个线程安全的数据结构，第二和第三个需要创建一个完整的并发程序，但不需要太大，同样，最后一个练习难度最大。

(1) 创建一个线程安全的切片类型，称之为safeSlice，并且具有可导出的SafeSlice接口：

```
type SafeSlice interface {  
    Append(interface{})          // 添加指定元素  
    At(int) interface{}          // 返回在指定位置的元素  
    Close() []interface{}        // 关闭通道并返回切片  
    Delete(int)                   // 删除指定位置元素  
    Len() int                     // 返回元素个数  
    Update(int, UpdateFunc)      // 更新指定位置的元素  
}
```

这也需要创建一个 safeSlice.run()方法，在里面创建一个底层的切片（类型为[]interface{}），还要有一个无限循环，遍历某个通道。还

必须要有一个New()函数，并在里面创建一个线程安全的切片，然后在一个独立的goroutine里执行safeSlice.run()方法。

线程安全的切片实现可以参考 7.2.3 节的安全映射实现，同样 safeSlice.Update() 方法会有死锁的风险。这道题的参考答案在 safeslice/safeslice.go 文件里，不到100 行代码。（可以使用apachereport4 程序来做测试，因为它使用了safeslice包。）

(2) 创建一个程序，从命令行接收一张或者多张图片的文件名，对于每个文件名，在控制台上打印一行HTML标签的字符串，格式如下：

```

```

这个程序应该使用固定数量的工作goroutine来并发处理这些图片，输出的顺序不重要（只要完整输出每一行就行），输出文件名时不要带上路径。命令行允许输入任意数量的文件，但是对于不是普通文件的或者不是图片的都忽略掉，还有所有的错误也会被忽略掉。

标准库里的 image.DecodeConfig() 函数能从 io.Reader（可由 os.Open() 返回）里得到一个图片的宽和高，不需要读取整个图像文件。这个函数可以辨别多种不同的图像格式，例如.jpg、.png等，但需要导入对应的包。但是我们并不直接使用这些包，所以为了避免Go编译器给我们一个“imported and not used”错误，我们需要将这些包导入为空标识符_，如_ "image/jpeg"、_ "image/png" 等。下一章我们会讨论这种类型的导入。

这本书的例子包括两种实现， imagetag1 是单线程的，没有通道，没有额外的goroutine，还有一个就是 imagetag2，这是并发的，类似于我们这章的cgrep2 程序， imagetag2/imagetag2.go 用了另一种大有不同的方法，大概 100 行代码。如果你只想关心并发方面的，你可以复制 imagetag1/imagetag1.go 文件到，例如， my_imagetag 目录，然后自己将它从单线程的转换成并发的。这也就需要你修改 main() 函数和 process()

函数，大概需要增加 40 行代码。Windows 用户需要使用 `commandLineFiles()` 函数来处理文件自动匹配。我们把两种实现都写好了，当然，有些很有信心的读者可能更倾向于从头写整个程序。

(3) 创建一个并发的程序，使用固定数量的工作 `goroutine`，从命令行接收一到多个 HTML 文件。程序分析每一个 HTML 文件，当找到一个标签时，就检查这个标签是否包含 `width` 和 `height` 属性字段，如果没有，就将相应缺少的补充进去。并发数据结构应该使用类似 `apachereport` 的方法来完成（但不是 `apachereport2` 版本），除非不需要结果通道。

因为 Go1 还没有 HTML 解析器，所以在这个练习里我们使用正则表达式 `regexp` 包和字符串的 `strings` 包 [4]。我们使用正则表达式 `<[iI][mM][gG][^>]+>` 来搜索图像里的所有的标签，还使用 `src=[""]([^\"]+)[\"]` 来标识标签里图片的文件名。（这些正则表达式不是很复杂，因为我们主要关注并发而不是正则表达式本身，你可以参考 3.6.5 节。）使用我们之前练习里讨论过的 `image.DecodeConfig()` 函数获得每个图像的大小。同样，Windows 用户需要 `commandLineFiles()` 来处理文件名自动匹配。

整个程序的并发结构是很容易理解的，但是处理逻辑则相当棘手。相对其他语言来说这是一个可喜的变化，因为在其他编程语言里并发问题的解决难度要远高于业务逻辑。

这个练习同样提供了两个参考答案，第一个是 `sizeimages1`（在文件 `sizeimages1/sizeimages1.go` 里），实现了所有它应该做的事情，但有一个缺陷是只有我们的程序和 HTML 页面在同一个目录下时才能查找到相应的图像文件。这是由于我们用 `regexp.Regexp.ReplaceAllStringFunc()` 函数将所有的标签更新了，这个方法需要一个签名为 `func(string) string` 的函数作为参数，前一个 `string` 是我们匹配到的字符串，后一个返回的 `string` 是我们需要用来替换的字符串。典型的例子就是，传入一个 `` 字符串。关键

是这个替换函数并不知道这个`splash.png`文件所在的路径，所以它假定这是在当前目录的，然后就有了我们这个限制，`sizeimages1`必须运行在HTML文件所在的目录。

我们可以尝试使用全局的`directory`字符串来解决这个问题，在使用替换函数之前，将当前HTML文件的路径保存到这个变量里去。但是这也不是在所有的场合下都适用。为什么呢？第二个参考答案是`sizeimages2`（在文件`sizeimages2/sizeimages2.go`里），使用一个闭包作为替换函数来捕获当前HTML文件的目录，然后，替换函数使用这个捕获到的目录作为图像的路径（要注意HTML标签里的图像可能是相对路径）。

这可能是这本书目前为止最有挑战性的例子了，需要查询`image`、`regexp`、`strings`等包里的函数。`sizeimages1.go`文件大概有160行代码，`sizeimages2.go`则不到170行。

[1].[有一些处理器号称拥有比实际更多的处理器，详见en.wikipedia.org/wiki/Hyperthreading。](http://en.wikipedia.org/wiki/Hyperthreading)

[2].[对于任意给定的数据块，如一个文件，SHA-1安全散列算法会产生一个20字节的值。相同的文件的SHA-1值是一样的，但不同的文件的SHA-1值总是不一样。](#)

[3].[本书撰写时，Go语言并没有对此功能的原生支持。](#)

[4].[有一个HTML解析器正在开发中，你阅读本书时可能就可用了。](#)

第8章 文件处理

在前面几章中我们看了几个与创建以及读写文件有关的例子。本章我们将深入了解一下Go语言中的文件处理，特别是如何读写标准格式（如 XML和JSON 格式）的文件以及自定义的纯文本和二进制格式文件。

由于前面的内容已覆盖 Go语言的所有特性（下一章将要讲到的使用自定义包和第三方包来创建程序的内容除外），现在我们可以灵活地使用Go语言提供的所有工具。我们会充分利用这种灵活性并利用闭包（参见5.6.3节）来避免重复性的代码，同时在某些情况下充分利用Go语言对面向对象的支持，特别是对为函数添加方法的支持。

本章的重点在于文件而非目录或者通用的文件系统。对于目录，前面章节的 `findduplicates` 示例（参见 7.2.5 节）展示了如何使用 `filepath.Walk()` 函数来迭代访问目录下的文件及其子目录。此外，标准库 `os` 包中的 `os.File` 类型提供了用于读取目录下的文件名的方法（`os.File.Readdirnames()`），以及用于获取目录下每一项的 `os.FileInfo` 值的方法（`os.File.Readdir()`）。

本章的第一节讲解了如何使用标准和自定义的文件格式进行文件的读写。第二节讲解了Go语言对处理压缩文件及相应的压缩算法的支持。

8.1 自定义数据文件

对一个程序非常普遍的需求包括维护内部数据结构，为数据交换提供导入导出功能，也支持使用外部工具来处理数据。由于我们这里的关注重点是文件处理，因此我们纯粹只关心如何从程序内部数据结构中读取数据并将其写入标准和自定义格式的文件中，以及如何从标准和自定义格式文件中读取数据并写入程序的内部数据结构中。

本节中，我们会为所有的例子使用相同的数据，以便直接比较不同的文件格式。所有的代码都来自 `invoicedata` 程序（在 `invoicedata` 目录中的 `invoicedata.go`、`gob.go`、`inv.go`、`jsn.go`、`txt.go` 和 `xml.go` 等文件中）。该程序接受两个文件名作为命令行参数，一个用于读，另一个用于写（它们必须是不同的文件）。程序从第一个文件中读取数据（以其后缀所表示的任何格式），并将数据写入第二个文件（也是以其后缀所表示的任何格式）。

由 `invoicedata` 程序创建的文件可跨平台使用，也就是说，无论是什么格式，Windows 上创建的文件都可在 Mac OS X 以及 Linux 上读取，反之亦然。Gzip 格式压缩的文件（如 `invoices.gob.gz`）可以无缝读写。压缩相关的内容在8.2节阐述。

这些数据由一个 `[]*Invoice` 组成，也就是说，是一个保存了指向 `Invoice` 值的指针的切片。每一个发票数据都保存在一个 `Invoice` 类型的值中，同时每一个发票数据都以 `[]*Item` 的形式保存着0个或者多个项。

<pre>type Invoice struct { Id int CustomerId int Raised time.Time Due time.Time Paid bool Note string Items []*Item }</pre>	<pre>type Item struct { Id string Price float64 Quantity int Note string }</pre>
--	---

这两个结构体用于保存数据。表8-1给出了一些非正式的对比，展示了每种格式下读写相同的50000份随机发票数据所需的时间，以及以

该格式所存储文件的大小。计时按秒计，并向上舍入到最近的十分之一秒。我们应该把计时结果认为是无绝对单位的，因为不同硬件以及不同负载情况下该值都不尽相同。大小一栏以千字节(KB)算，该值在所有机器上应该都是相同的。对于该数据集，虽然未压缩文件的大小千差万别，但压缩文件的大小都惊人的相似。而代码的函数不包括所有格式通用的代码（例如，那些用于压缩和解压缩以及定义结构体的代码）。

表8-1 各种格式的速度以及大小对比

后缀	读取	写入	大小 (KiB)	读/写 LOC	格式
.gob	0.3	0.2	7 948	21 + 11 = 32	Go 二进制
.gob.gz	0.5	1.5	2 589		
.json	4.5	2.2	16 283	32 + 17 = 49	JSON
.json.gz	4.5	3.4	2 678		
.xml	6.7	1.2	18 917	45 + 30 = 75	XML
.xml.gz	6.9	2.7	2 730		
.txt	1.9	1.0	12 375	86 + 53 = 139	纯文本 (UTF-8)
.txt.gz	2.2	2.2	2 514		
.inv	1.7	3.5	7 250	128 + 87 = 215	自定义二进制
.inv.gz	1.6	2.6	2 400		

这些读写时间和文件大小在我们的合理预期范围内，除了纯文本格式的读写异常快之外。这得益于 `fmt` 包优秀的打印和扫描函数，以及我们设计的易于解析的自定义文本格式。对于JSON和XML 格式，我们只简单地存储了日期部分而非存储默认的`time.Time` 值（一个ISO-8601 日期/时间字符串），通过牺牲一些速度和增加一些额外代码稍微减小了文件的大小。例如，如果让JSON代码自己来处理`time.Time`值，它能够运行得更快，并且其代码行数与Go语言二进制编码差不多。

对于二进制数据，Go语言的二进制格式是最便于使用的。它非常快且极端紧凑，所需的代码非常少，并且相对容易适应数据的变化。然而，如果我们使用的自定义类型不原生支持 `gob`编码，我们必须让该

类型满足`gob.Encoder`和`gob.Decoder`接口，这样会导致`gob`格式的读写相当得慢，并且文件大小也会膨胀。

对于可读的数据，**XML**可能是最好使用的格式，特别是作为一种数据交换格式时非常有用。与处理**JSON**格式相比，处理**XML**格式需要更多行代码。这是因为Go 1没有一个`xml.Marshaler`接口（这个缺失有希望在Go 1.x之后的发行版中得到弥补），也因为我们这里使用了并行的数据类型（`XMLInvoice`和`XMLItem`）来帮助映射**XML**数据和发票数据（`Invoice`和`Item`）。使用**XML**作为外部存储格式的应用程序可能不需要并行的数据类型或者也不需要 `invoicedata` 程序这样的转换，因此就有可能比`invoicedata`例子中所给出的更快，并且所需的代码也更少。

除了读写速度和文件大小以及代码行数之外，还有另一个问题值得考虑：格式的稳健性。例如，如果我们为`Invoice`结构体和`Item`结构体添加了一个字段，那么就必须再改变文件的格式。我们的代码适应读写新格式并继续支持读旧格式的难易程度如何？如果我们为文件格式定义版本，这样的变化就很容易被适应（会以本章一个练习的形式给出），除了让**JSON**格式同时适应读写新旧格式稍微复杂一点之外。

除了`Invoice`和`Item`结构体之外，所有文件格式都共享以下常量：

```
const (  
    fileType          = "INVOICES"           // 用于纯文本格式  
    magicNumber       = 0x125D               // 用于二进制格式  
    fileVersion       = 100                  // 用于所有的格式  
    dataFormat        = "2006-01-02"        // 必须总是使用该日期  
)
```

`magicNumber`用于唯一标记发票文件 [1] 。 `fileVersion`用于标记发票文件的版本，该标记便于之后修改程序来适应数据格式的改变。 `dataFormat`稍后介绍（参见5.1.1.2节），它表示我们希望数据如何按照可读的格式进行格式化。

同时，我们也创建了一对接口。

```
type InvoiceMarshaler interface {  
    MarshalInvoices(writer io.Writer, invoices []*Invoice) error  
}  
  
type InvoiceUnmarshaler interface {  
    UnmarshalInvoices(reader io.Reader) ([]*Invoice, error)  
}
```

这样做的目的是以统一的方式针对特定格式使用 `reader`和`writer`。例如，下列函数是`invoicedata`程序用来从一个打开的文件中读取发票数据的。

```
func readInvoices(reader io.Reader, suffix string)( []*Invoice, error) {  
    var unmarshaler InvoicesUnmarshaler  
    switch suffix {  
    case ".gob":  
        unmarshaler = GobMarshaler{}  
    case ".inv":  
        unmarshaler = InvMarshaler{}  
    case ".jsn", ".json":  
        unmarshaler = JSONMarshaler{}  
    case ".txt":  
        unmarshaler = TxtMarshaler{}  
    case ".xml":  
        unmarshaler = XMLMarshaler{}  
    }
```

```

    }
    if unmarshaler != nil {
        return unmarshaler.UnmarshalInvoices(reader)
    }
    return nil, fmt.Errorf("unrecognized input suffix: %s", suffix)
}

```

其中，`reader` 是任何能够满足 `io.Reader` 接口的值，例如，一个打开的文件（其类型为 `*os.File`）、一个 `gzip` 解码器（其类型为 `*gzip.Reader`）或者一个 `string.Reader`。字符串 `suffix` 是文件的后缀名（从 `.gz` 文件中解压之后）。在接下来的小节中我们将会看到 `GobMarshaler` 和 `InvMarshaler` 等自定义的类型，它们提供了 `MarshalInvoices()` 和 `UnmarshalInvoices()` 方法（因此满足 `InvoicesMarshaler` 和 `InvoicesUnmarshaler` 接口）。

8.1.1 处理JSON文件

根据 www.json.org 介绍，JSON（JavaScript对象表示法，JavaScript Object Notation）是一种易于人读写并且易于机器解析和生成的轻量级的数据交换格式。JSON 是一种使用 UTF-8 编码的纯文本格式。由于写起来比 XML 格式方便，并且（通常）更为紧凑，而所需的处理时间也更少，JSON 格式已经越来越流行，特别是在通过网络连接传送数据方面。

这里是一个简单的发票数据的JSON表示，但是它省略了该发票的第二项的大部分字段。

```

{
    "Id": 4461,
    "CustomerId": 917,

```

```

    "Raised": "2012-07-22",
    "Due": "2012-08-21",
    "Paid": true,
    "Note": "Use trade entrance",
    "Items": [
        {
            "Id": "AM2574",
            "Price": 415.8,
            "Quantity": 5,
            "Note": ""
        },
        {
            "Id": "MI7296",
            ...
        }
    ]
}

```

通常，`encoding/json` 包所写的JSON 数据没有任何不必要的空格，但是这里我们为了更容易看明白数据的结构而使用了缩进和空白来展示它。虽然 `encoding/json` 包支持`time.Time`s，但是我们通过自己实现自定义的`MarshalJSON()`和`UnmarshalJSON()` `Invoice`方法来处理发票的开具和到期日期。这样我们就可以存储更短的日期字符串（因为对于我们的数据来说，其时间部分始终为0），例如“2012-09-06”，而非整个日期/时间值，如“2012-09-06T00:00:00Z”。

8.1.1.1 写JSON文件

我们创建了一个基于空结构体的类型，它定义了与JSON 相关的`MarshalInvoices()`和`UnmarshalInvoices()`方法。

```
type JSONMarshaler struct{}
```

该类型满足我们在前文看到的 `InvoicesMarshaler` 和 `InvoicesUnmarshaler`接口（见8.1节）。

这里的方法使用`encoding/json`包中标准的Go到JSON序列化函数将`[]*Invoice`项中的所有数据以JSON格式写入一个`io.Writer`中。该`writer`可以是`os.Create()`函数返回的`*os.File`，或者是 `gzip.NewWriter()`函数返回的`*gzip.Writer`，或者是任何满足`io.Writer`接口的其他值。

```
unc (JSONMarshaler) MarshalInvoices(writer io.Writer, invoices
[]*Invoice) error {
    encoder := json.NewEncoder(writer)
    if err := encoder.Encode(fileType); err != nil {
        return err
    }
    if err := encoder.Encode(fileVersion); err != nil {
        return err
    }
    return encoder.Encode(invoices)
}
```

`JSONMarshaler`类型没有数据，因此我们没必要将其值赋值给一个接收器变量。

函数开始处，我们创建了一个JSON编码器，它包装了`io.Writer`，可以接收我们写入的支持JSON编码的数据。

我们使用`json.Encoder.Encode()`方法来写入数据。该方法能够完美地处理发票切片，其中每个发票都包含一到多个项的切片。该方法返回一个错误值或者空值`nil`。如果返回的是一个错误值，则立即返回给调用者。

文件的类型和版本不是必须写入的，但在后面一个练习中会看到，这样做是为了以后更容易更改文件格式（例如，为了适应Invoice和Item结构体中额外的字段），以及为了能够同时支持读取新旧格式的数据。

需注意的是，该方法实际上与它所编码的数据类型无关，因此很容易创建类似函数用于写入其他可JSON编码的数据。另外，只要新的文件格式中新增的字段是导出的且支持JSON编码，该JSONMarshaler.MarshalInvoices()方法无需做任何更改。

如果这里所给出的代码就是JSON相关代码的全部，这样当然可以很好地工作。然而，由于我们希望更好地控制JSON的输出，特别是对time.Time值的格式化，我们还为Invoice类型提供了一个满足json.Marshaler接口的MarshalJSON()方法。json.Encode()函数足够智能，它会去检查所需编码的值是否支持json.Marshaler接口，如果支持，该函数会使用该值的MarshalJSON()方法而非内置的编码代码。

```
type JSONInvoice struct {
    Id          int
    CustomerId int
    Raised      string    // Invoice结构体中的time.Time
    Due         string    // Invoice结构体中的time.Time
    Paid        bool
    Note        string
    Items       []*Item
}

func (invoice Invoice) MarshalJSON()([]byte, error) {
    jsonInvoice := JSONInvoice {
        invoice.Id,
        invoice.CustomerId,
```

```

        invoice.Raised.Format(dateFormat),
        invoice.Due.Format(dateFormat),
        invoice.Paid,
        invoice.Note,
        invoice.Items,
    }
    return json.Marshal(jsonInvoice)
}

```

该自定义的`Invoice.MarshalJSON()`方法接受一个已有的`Invoice` 值，返回一个该数据 `JSON` 编码后的版本。该函数的第一个语句简单地将发票的各个字段复制到自定义的`JSONInvoice`结构体中，同时将两个`time.Time`值转换成字符串。由于`JSONInvoice`结构体的字段都是布尔类型、数字或者字符串，该结构体可以使用 `json.Marshal()`函数进行编码，因此我们使用该函数来完成工作。

为了将日期/时间（即 `time.Time` 值）以字符串的形式写入，我们必须使用 `time.Time.Format()`方法。该方法接受一个格式字符串，它表示该日期/时间值应该如何写入。该格式字符串非常特殊，必须是一个 Unix 时间 1 136 243 045 的字符串表示，即精确的日期/时间值 2006-01-02T15:04:05Z07:00，或者跟这里一样，使用该日期/时间值的子集。该特殊的日期/时间值是任意的，但必须是确定的，因为没有其他的值来声明日期、时间以及日期/时间的格式。

如果我们想自定义日期/时间格式，它们必须按照 Go 语言的日期/时间格式来写。假如我们要以星期、月、日和年的形式来写日期，我们必须使用“Mon, Jan 02, 2006”这种格式，或者如果我们希望删除前导的 0，就必须使用“Mon, Jan _2, 2006”这种格式。`time`包的文档中有完整的描述，并列出了一些预定义的格式字符串。

8.1.1.2 读JSON文件

读JSON数据与写JSON数据一样简单，特别是当将数据读回与写数据时类型一样的变量时。JSONMarshaler.UnmarshalInvoices()方法接受一个 io.Reader 值，该值可以是一个 os.Open()函数返回的*os.File 值，或者是一个 gzip.NewReader()函数返回的*gzip.Reader值，也可以是任何满足io.Reader接口的值。

```
func (JSONMarshaler) UnmarshalInvoices(reader io.Reader)
([]*Invoice, error){
    decoder := json.NewDecoder(reader)
    var kind string
    if err := decoder.Decode(&kind); err != nil {
        return nil, err
    }
    if kind != fileType {
        return nil, errors.New("Cannot read non-invoices json file")
    }
    var version int
    if err := decoder.Decode(&version); err != nil {
        return nil, err
    }
    if version > fileVersion {
        return nil, fmt.Errorf("version %d is too new to read", version)
    }
    var invoices []*Invoice
    err := decoder.Decode(&invoices)
    return invoices, err
}
```

我们需读入3项数据：文件类型、文件版本以及完整的发票数据。`json.Decoder.Decode()`方法接受一个指针，该指针所指向的值用于存储解码后的JSON数据，解码后返回一个错误值或者`nil`。我们使用前两个变量（`kind`和`version`）来保证接受一个JSON格式的发票文件，并且该文件的版本是我们能够处理的。然后程序读取发票数据，在该过程中，随着`json.Decoder.Decode()`方法所读取发票数的增多，它会增加`invoices`切片的长度，并将相应发票的指针（及其项目）保存在切片中，这些指针是`UnmarshalInvoices`函数在必要时实时创建的。最后，该方法返回解码后的发票数据和一个`nil`值。或者，如果解码过程中遇到了问题则返回一个`nil`值和一个错误值。

如果我们之前纯粹依赖于`json`包内置的功能把数据的创建及到期日期按照默认的方式序列化，那么这里给出的代码已经足以反序列化一个JSON格式的发票文件。然而，由于我们使用自定义的方式来序列化数据的建立和到期日期`time.Time`（只存储日期部分），我们必须提供一个自定义的反序列化方法，该方法理解我们的自定义序列化流程。

```
func (invoice *Invoice) UnmarshalJSON(data []byte) (err error) {
    var jsonInvoice JSONInvoice
    if err = json.Unmarshal(data, &jsonInvoice); err != nil {
        return err
    }
    var raised, due time.Time
    if raised, err = time.Parse(dateFormat, jsonInvoice.Raised);
        err != nil {
            return err
        }
    if due, err = time.Parse(dateFormat, jsonInvoice.Due); err != nil {
```

```

        return err
    }
    *invoice = Invoice {
        jsonInvoice.Id,
        jsonInvoice.CustomerId,
        raised,
        due,
        jsonInvoice.Paid,
        jsonInvoice.Note,
        jsonInvoice.Items,
    }
    return nil
}

```

该方法使用与前面一样的JSONInvoice结构体，并且依赖于json.Unmarshal()函数来填充数据。然后，我们将反序列化后的数据以及转换成time.Time的日期值赋给新创建的Invoice变量。

json.Decoder.Decode()足够智能会检查它需要解码的值是否满足json.Unmarshaler接口，如果满足则使用该值自己的UnmarshalJSON()方法。

如果发票数据因为新添加了导出字段而发生改变，该方法能继续正常工作的前提是我们必须让Invoice.UnmarshalJSON()方法也能处理版本变化。另外，如果新添加字段的零值不可被接受，那么当以原始格式读文件的时候，我们必须对数据做一些后期处理，并给它们一个合理的值。（有一个练习需要添加新字段以及进行此类后期处理工作。）

虽然要支持两个或者更多个版本的JSON文件格式有点麻烦，但JSON是一种很容易处理的格式，特别是如果我们创建的结构体的导出

字段比较合理时。同时，`json.Encoder.Encode()` 函数和 `json.Decoder.Decode()` 函数也不是完美可逆的，这意味着序列化后得到的数据经过反序列化后不一定能够得到原始的数据。因此，我们必须小心检查，保证它们对我们的数据有效。

顺便提一下，还有一种叫做BSON（Binary JSON）的格式与JSON非常类似，它比JSON更为紧凑，并且读写速度也更快。godashboard.appspot.com/project 网页上有一个支持BSON格式的第三方包（`gobson`）。（安装和使用第三方包的内容将在第9章阐述。）

8.1.2 处理XML文件

XML（eXtensible Markup Language）格式被广泛用作一种数据交换格式，并且自成一种文件格式。与JSON相比，XML复杂得多，手动写起来也啰嗦而且乏味得多。

`encoding/xml`包可以用在结构体和XML格式之间进行编解码，其方式跟`encoding/json`包类似。然而，与`encoding/json`包相比，XML的编码和解码在功能上更苛刻得多。这部分是由于`encoding/xml`包要求结构体的字段包含格式合理的标签（然而JSON格式却不需要）。同时，Go 1的`encoding/xml`包没有`xml.Marshaler`接口，因此与编解码JSON格式和Go语言的二进制格式相比，我们处理XML格式时必须写更多的代码。（该问题有望在Go 1.x发行版中得以解决。）

这里有个简单的XML格式的发票文件。为了适应页面的宽度和容易阅读，我们添加了换行和额外的空白。

```
<INVOICE   Id="2640"   CustomerId="968"   Raised="2012-08-27"
Due="2012-09-26"
      Paid="false"><NOTE>See           special           Terms           &amp;
Conditions</NOTE>
```

```

    <ITEM Id="MI2419" Price="342.80" Quantity="1"><NOTE>
</NOTE></ITEM>
    <ITEM Id="OU5941" Price="448.99" Quantity="3"><NOTE>
    &quot;Blue&quot; ordered but will accept &quot;Navy&quot;
</NOTE> </ITEM>
    <ITEM Id="IF9284" Price="475.01" Quantity="1"><NOTE>
</NOTE></ITEM>
    <ITEM Id="TI4394" Price="417.79" Quantity="2"><NOTE>
</NOTE></ITEM>
    <ITEM Id="VG4325" Price="80.67" Quantity="5"><NOTE>
</NOTE></ITEM>
</INVOICE>

```

对于xml包中的编码器和解码器而言，标签中如果包含原始字符数据（如invoice和item中的Note字段）处理起来比较麻烦，因此invoicedata示例使用了显式的<NOTE>标签。

8.1.2.1 写XML文件

encoidng/xml包要求我们使用的结构体中的字段包含encoding/xml包中所声明的标签，所以我们不能直接将Invoice和Item结构体用于XML序列化。因此，我们创建了针对XML格式的XMLInvoices、XMLInvoice和XMLItem结构体来解决这个问题。同时，由于invoicedata程序要求我们有并行的结构体集合，因此必须提供一种方式来让它们相互转换。当然，使用XML格式作为主要存储格式的应用程序只需一个结构体（或者一个结构体集合），同时要将必要的encoidng/xml包的标签直接添加到结构体的字段中。

下面是保存整个数据集合的XMLInvoices结构体。

```

type XMLInvoices struct {
    XMLName xml.Name          `xml:"INVOICES"`

```

```

    Version    int           `xml:"version,attr"`
    Invoice     []*XMLInvoice `xml:"INVOICE"`
}

```

在Go语言中，结构体的标签本质上没有任何语义，它们只是可以使用Go语言的反射接口获得的字符串（参见9.4.9节）。然而，`encoding/xml`包要求我们使用该标签来提供如何将结构体的字段映射到XML的信息。`xml.Name`字段用于为XML中的标签命名，该标签包含了该字段所在的结构体。以``xml:"",attr"``标记的字段将成为该标签的属性，字段名字将成为属性名。我们也可以根据自己的喜好使用另一个名字，只需在所给的名字签名加上一个逗号。这里，我们把**Version**字段当做一个叫做**version**的属性，而非默认的名字**Version**。如果标签只包含一个名字，则该名字用于表示嵌套的标签，如此例中的**<INVOICE>**标签。有一个非常重要的细节需注意的是，我们把XMLInvoices的发票字段命名为**Invoice**，而非**Invoices**，这是为了匹配XML格式中的标签名（不区分大小写）。

下面是原始的**Invoice**结构体，以及与XML格式相对应的XMLInvoice结构体。

<pre> type Invoice struct { Id int CustomerId int Raised time.Time Due time.Time Paid bool Note string Items []*Item } </pre>	<pre> type XMLInvoice struct { XMLName xml.Name `xml:"INVOICE"` Id int `xml:"",attr` CustomerId int `xml:"",attr` Raised string `xml:"",attr` Due string `xml:"",attr` Paid bool `xml:"",attr` Note string `xml:"NOTE"` Item []*XMLItem `xml:"ITEM"` } </pre>
--	---

在这里，我们为属性提供了默认的名字。例如，字段**CustomerId**在XML中对应一个属性，其名字与该字段的名称完全一样。这里有两个可嵌套的标签：**<NOTE>**和**<ITEM>**，并且如XMLInvoices结构体一

样，我们把XMLInvoice的item字段定义成Item（大小写不敏感）而非Items，以匹配标签名。

由于我们希望自己处理创建和到期日期（只存储日期），而非让encoding/xml包来保存完整的日期/时间字符串，我们为它们在XMLInvoice结构体中定义了相应的Raised和Due字段。

下面是原始的Item结构体，以及与XML相对应的XMLItem结构体。

<pre>type Item struct { Id string Price float64 Quantity int Note string }</pre>	<pre>type XMLItem struct { XMLName xml.Name 'xml:"ITEM"' Id string 'xml:",attr"' Price float64 'xml:",attr"' Quantity int 'xml:",attr"' Note string 'xml:"NOTE"' }</pre>
--	---

除了作为嵌套的<NOTE>标签的Note字段和用于保存该XML标签名的XMLName字段之外，XMLItem的字段都被打上了标签以作为属性。

正如处理JSON格式时所做的那样，对于XML格式，我们创建了一个空的结构体并关联了XML相关的MarshalInvoices()方法和UnmarshalInvoices()方法。

```
type XMLMarshaler struct{}
```

该类型满足前文所述的InvoicesMarshaler和InvoiceUnmarshaler接口（参见8.1节）。

```
func (XMLMarshaler) MarshalInvoices(writer io.Writer, invoices  
[]*Invoice) error {  
    if _, err := writer.Writer([]byte(xml.Header)); err != nil {  
        return err  
    }  
    xmlInvoices := XMLInvoicesForInvoices(invoices)
```

```

    encoder := xml.NewEncoder(writer)
    return encoder.Encode(xmlInvoices)
}

```

该方法接受一个`io.Writer`(也就是说, 任何满足`io.Writer`接口的值如打开的文件或者打开的压缩文件), 以用于写入XML数据。该方法从写入标准的XML头部开始(该`xml.Header`常量的末尾包含一个新行)。然后, 它将所有的发票数据及其项写入相应的XML结构体中。这样做虽然看起来会耗费与原始数据相同的内存, 但是由于Go语言的字符串是不可变的, 因此在底层只将原始数据字符串的引用复制到XML结构体中, 因此其代价并不是我们所看到的那么大。而对于直接使用带有XML标签的结构体的应用而言, 其数据没必要再次转换。

一旦填充好 `xmlInvoices` (其类型为 `XMLInvoices`) 后, 我们创建了一个新的`xml.Encoder`, 并将我们希望写入数据的`io.Writer`传给它。然后, 我们将数据编码成XML格式, 并返回编码器的返回值, 该值可能为一个`error`值也可能为`nil`。

```

func XMLInvoicesForInvoices(invoices []*Invoice) *XMLInvoices {
    xmlInvoices := &XMLInvoices{
        Version: fileVersion,
        Invoice: make([]*XMLInvoice, 0, len(invoices)),
    }
    for _, invoice := range invoices {
        xmlInvoices.Invoice = append(xmlInvoices.Invoice,
            XMLInvoiceForInvoice (invoice))
    }
    return xmlInvoices
}

```

该函数接受一个[]*Invoice 值并返回一个 *XMLInvoices 值，其中包含转换成*XMLInvoices（还包含 *XMLItems 而非 *Items）的所有数据。该函数又依赖于XmlInvoiceForInvoice()函数来为其完成所有工作。

我们不必手动填充 `xml.Name` 字段（除非我们想使用名字空间），因此在这里，当创建 `*XMLInvoices` 的时候，我们只需填充 `Version` 字段以保证我们的标签有一个 `version` 属性，例如 `<INVILES version="100">`。同时，我们将 `Invoice` 字段设置成一个空间足够容纳所有的发票数据的空切片。这样做不是严格必须的，但是与将该字段的初始值留空相比，这样做可能更高效，因为这样做意味着调用内置的 `append()` 函数时无需分配内存和复制数据以扩充切片容量。

```
func XMLInvoiceForInvoice(invoice *Invoice) *XMLInvoice {
    xmlInvoice := &XMLInvoice{
        Id:            invoice.id,
        CustomerId:    invoice.CustomerId,
        Raised:        invoice.Raised.Format(dateFormat),
        Due:            invoice.Due.Format(dateFormat),
        Paid:          invoice.Paid,
        Note:          invoice.Note,
        Item:          make([]*XMLItem, 0, len(invoice.Items)),
    }

    for _, item := range invoice.Items {
        xmlItem := &XMLItem {
            Id:        item.Id,
            Price:     item.Price,
            Quantity:  item.Quantity,
            Note:      item.Note,
        }
    }
}
```

```

        xmlInvoice.Item = append(xmlInvoice.Item, xmlItem)
    }
    return xmlInvoice
}

```

该函数接受一个 `Invoice` 值并返回一个等价的 `XMLInvoice` 值。该转换非常直接,只需简单地将 `Invoice` 中每个字段的值复制至 `XMLInvoice` 字段中。由于我们选择自己来处理创建以及到期日期（因此我们只需存储日期而非完整的日期/时间），我们只需将其转换成字符串。而对于 `Invoice.Items` 字段，我们将每一项转换成 `XMLItem` 后添加到 `XMLInvoice.Item` 切片中。与前面一样，我们使用相同的优化方式，创建 `Item` 切片时分配了足够多的空间以避免 `append()` 时需要分配内存和复制数据。前文阐述 `JSON` 格式时我们已讨论过 `time.Time` 值的写入（参见8.1.1.1节）。

最后需要注意的是，我们的代码中没有做任何 `XML` 转义，它是由 `xml.Encoder.Encode()` 方法自动完成的。

8.1.2.2 读XML文件

读XML文件比写XML文件稍微复杂，特别是在必须处理一些我们自定义的字段的时候（例如日期）。但是，如果我们使用合理的打上XML标签的结构体，就不会复杂。

```

func (XMLMarshaler) UnmarshalInvoices(reader io.Reader)
([]*Invoice, error) {
    xmlInvoices := &XMLInvoices{}
    decoder := xml.NewDecoder(reader)
    if err := decoder.Decode(xmlInvoices); err != nil {
        return nil, err
    }
    if xmlInvoices.Version > fileVersion {

```

```

        return nil, fmt.Errorf("version %d is too new to read",
xmlInvoices.Version)
    }
    return xmlInvoices.Invoices()
}

```

该方法接受一个 `io.Reader`（也就是说，任何满足 `io.Reader` 接口的值如打开的文件或者打开的压缩文件），并从其中读取XML。该方法的开始处创建了一个指向空XMLInvoices结构体的指针，以及一个 `xml.Decoder` 用于读取 `io.Reader`。然后，整个XML文件由 `xml.Decoder.Decode()`方法解析，如果解析成功则将XML文件的数据填充到该*XMLInvoices结构体中。如果解析失败（例如，XML文件语法有误，或者该文件不是一个合法的发票文件），那么解码器会立即返回错误值给调用者。如果解析成功，我们再检查其版本，如果该版本是我们能够处理的，就将该XML结构体转换成我们程序内部使用的结构体。当然，如果我们直接使用带XML标签的结构体，该转换步骤就没必要了。

```

func (xmlInvoices *XMLInvoices) Invoices() (invoices []*Invoice, err
error){
    invoices = make([]*Invoice, 0, len(xmlInvoices.Invoice))
    for _, XMLInvoice := range xmlInvoices.Invoice {
        invoice, err := xmlInvoice.Invoice()
        if err != nil {
            return nil, err
        }
        invoices = append(invoices, invoice)
    }
    return invoices, nil
}

```

```
}
```

该XMLInvoices.Invoices()方法将一个*XMLInvoices值转换成一个[]*Invoice值，它是 XmlInvoicesForInvoices()函数的逆反操作，并将具体的转换工作交给XMLInvoice.Invoice()方法完成。

```
func (xmlInvoice *XMLInvoice) Invoice() (invoice *Invoice, err error)
{
    invoice = &Invoice{
        Id:          xmlInvoice.Id,
        CustomerId:  xmlInvoice.CustomerId,
        Paid:         xmlInvoice.Paid,
        Note:        strings.TrimSpace(xmlInvoice.Note),
        Items:       make([]*Item, 0, len(xmlInvoice.Item)),
    }
    if invoice.Raised, err = time.Parse(dateFormat, xmlInvoice.Raised);
        err != nil {
        return nil, err
    }
    if invoice.Due, err = time.Parse(dateFormat, xmlInvoice.Due);
        err != nil{
        return nil, err
    }
    for _, xmlItem := range xmlInvoice.Item {
        item := &Item {
            Id:          xmlItem.Id,
            Price:       xmlItem.Price,
            Quantity:    xmlItem.Quantity,
            Note:        strings.TrimSpace(xmlItem.Note),
```

```

    }
    invoice.Items = append(invoice.Items, item)
}
return invoice, nil
}

```

该方法用于返回与调用它的*XMLInvoice值相应的*Invoice值。

该方法在开始处创建了一个Invoice值，其大部分字段都由来自XMLInvoice的数据填充，而Items字段则设置成一个容量足够大的空切片。

然后，由于我们选择自己处理这些，因此手动填充两个日期/时间字段。time.Parse()函数接受一个日期/时间格式的字符串（如前所述，该字符串必须基于精确的日期/时间值，如2006-01-02T15:04:05Z07:00），以及一个需要解析的字符串，并返回等价的time.Time值和nil，或者，返回一个nil和一个错误值。

接下来是填充发票的Items字段，这是通过迭代XMLInvoice的Item字段中的*XMLItems并创建相应的*Items来完成的。最后，返回*Invoice。

正如写XML时一样，我们无需关心对所读取的XML数据进行转义，xml.Decoder.Decode()函数会自动处理这些。

xml包支持比我们这里所需的更为复杂的标签，包括嵌套。例如，标签名为`xml:"Books> Author"`产生的是<Books><Author>content</Author></Books>这样的XML内容。同时，除了`xml:","attr"`之外，该包还支持`xml:","chardata"`这样的标签表示将该字段当做字符数据来写，支持`xml:","innerxml"`这样的标签表示按照字面量来写该字段，以及`xml:","comment"`这样的标签表示将该字段当做XML注释。因此，通过使用标签化的结构体，我们可以充分利用好这些方便的编码解码函数，同时合理控制如何读写XML数据。

8.1.3 处理纯文本文件

对于纯文本文件，我们必须创建自定义的格式，理想的格式应该易于解析和扩展。

下面是某单个发票以自定义纯文本格式存储的数据。

```
INVOICE    ID=5441    CUSTOMER=960    RAISED=2012-09-06
DUE=2012-10-06 PAID=true

ITEM ID=BE9066 PRICE=400.89 QUANTITY=7: Keep out of
<direct> sunlight
ITEM ID=AM7240 PRICE=183.69 QUANTITY=2
ITEM ID=PT9110 PRICE=105.40 QUANTITY=3: Flammable
```



在该格式中，每个发票是一个INVOICE行，然后是一个或者多个ITEM行，最后是换页符。每一行（无论是发票还是它们的项）的基本结构都相同：起始处有一个单词表示该行的类型，接下来是一个空格分隔的“键=值”序列，以及可选的跟在一个冒号和一个空格后面的注释文本。

8.1.3.1 写纯文本文件

由于Go语言的fmt包中打印函数强大而灵活（这在前文已有阐述，详见3.5节），写纯文本数据非常简单直接。

```
type TxtMarshaler struct{}

func (TxtMarshaler) MarshalInvoices(writer io.Writer,
    invoices []*Invoice) error {
    bufferedWriter := bufio.NewWriter(writer)
    defer bufferedWriter.Flush()
    var write writerFunc = func(format string, args...interface{}) error {
        _, err := fmt.Fprintf(bufferedWriter, format, args...)
    }
```



```

        return err
    }
    if err := write("%s %d\n", fileType, fileVersion); err != nil {
        return err
    }
    for _, invoice := range invoices {
        if err := write.WriteInvoice(invoice); err != nil {
            return err
        }
    }
    return nil
}

```

该方法在开始处创建了一个带缓冲区的**writer**，用于操作所传入的文件。延迟执行刷新缓冲区的操作是必要的，这可以保证我们所写的的数据确实能够写入文件（除非发生错误）。

与以**if _, err := fmt.Fprintf(bufferedWriter,...); err != nil {return err}**的形式来检查每次写操作不同的是，我们创建了一个函数字面量来做两方面的简化。第一，该**writer()**函数会忽略**fmt.Fprintf()**函数报告的所写字节数。其次，该函数处理了**bufferedWriter**，因此我们不必在自己的代码中显式地提到。

我们本可以将 **write()** 函数传给辅助函数的，例如，**writeInvoice(write, invoice)**。但不同于此做法的是，我们往前更进了一步，将该方法添加到**writerFunc**类型中。这是通过声明接受一个**writerFunc**值作为其接收器的方法（即函数）来达到，跟定义任何其他类型一样。这样就允许我们以**write.writeInvoice(invoice)**这样的形式调用，也就是说，在**write()**函数自身上调用方法。并且，由于这些方法接受**write()**函数作为它们的接收器，我们就可以使用**write()**函数。

需注意的是，我们必须显式地声明 `write()` 函数的类型（`writerFunc`）。如果不这样做，Go 语言就会将其类型定义为 `func(string,...interface{}) error`（当然，它本来就是这种类型），并且不允许我们在其上调用 `writerFunc` 方法（除非我们使用类型转换的方法将其转换成 `writerFunc` 类型）。

有了方便的 `write()` 函数（及其方法），我们就可以开始写入文件类型和文件版本（后者使得容易适应数据的改变）。然后，我们迭代每一个发票项，针对每一次迭代，我们调用 `write()` 函数的 `writeInvoice()` 方法。

```
const noteSep = ":"
type writerFunc func(string,..interface{}) error
func (write writerFunc) writeInvoice(invoice *Invoice) error {
    note := ""
    if invoice.Note != "" {
        note = noteSep + " " + invoice.Note
    }
    if err := write("INVOICE ID=%d CUSTOMER=%d RAISED=%s
DUE=%s" +
        "PAID=%t%s\n", invoice.Id, invoice.CustomerId,
        invoice.Raised.Format(dateFormat),
        invoice.Due.Format(dateFormat), invoice.Paid, note); err != nil {
    return err
    }
    if err := write.writeItems(invoice.Items); err != nil {
        return err
    }
    return write("\f\n")
}
```

```
}
```

该方法用于写每一个发票项。它接受一个要写的发票项，同时使用作为接收器传入的`write()`函数来写数据。

发票数据一次性就可以写入。如果给出了注释文本，我们就在其前面加入冒号以及空格来将其写入。对于日期/时间（即 `time.Time` 值），我们使用 `time.Time.Format()` 方法，跟我们以JSON和XML格式写入数据时一样。而对于布尔值，我们使用`%t`格式指令，也可以使用`%v`格式指令或`strconv.FormatBool()`函数。

一旦发票行写好了，就开始写发票项。最后，我们写入分页符和一个换行符，表示发票数据的结束。

```
func (write writerFunc) writeItems(items []*Item) error {
    for _, item := range items {
        note := ""
        if item.Note != "" {
            note = noteSep + " " + item.Note
        }
        if err := write("ITEM    ID=%s    PRICE=%.2f
QUANTITY=%d%s\n", item.Id,
            item.Price, item.Quantity, note); err != nil {
            return err
        }
    }
    return nil
}
```

该`writeItems()`方法接受发票的发票项，并使用作为接收器传入的`write()`函数来写数据。它迭代每一个发票项并将其写入，并且也跟写入发票数据一样，如果其注释文档为空则无需写入。

8.1.3.2 读纯文本文件

打开并读取一个纯文本格式的数据跟写入纯文本格式数据一样简单。要解析文本来重建原始数据可能稍微复杂，这需根据格式的复杂性而定。

有4种方法可以使用。前3种方法包括将每行切分，然后针对非字符串的字段使用转换函数如`strconv.Atoi()`和`time.Parse()`。这些方法是：第一，手动解析（例如，一个字母一个字母或者一个字一个字地解析），这样做实现起来烦琐，不够健壮并且也慢；第二，使用`fmt.Fields()`或者`fmt.Split()`函数来将每行切分；第三，使用正则表达式。对于该`invoicedata`程序，我们使用第四种方法。无需将每行切分或者使用转换函数，因为我们所需的功能都能够交由`fmt`包的扫描函数处理。

```
func (TxtMarshaler) UnmarshalInvoices(reader io.Reader) ([]*Invoice,
error) {
    bufferedReader := bufio.NewReader(reader)
    if err := checkTxtVersion(bufferedReader); err != nil {
        return nil, err
    }
    var invoices []*Invoice
    eof := false
    for lino := 2; !eof; lino++ {
        line, err := bufferedReader.ReadString('\n')
        if err == io.EOF {
            err = nil           // io.EOF不是一个真正的错误
            eof = true         // 下一次迭代的时候会终止循环
        } else if err != nil {
            return nil, err    // 遇到真正的错误则立即停止
        }
    }
}
```

```

    }
    if invoices, err = parseTxtLine(lino, line, invoices); err != nil {
        return nil, err
    }
}
return invoices, nil
}

```

针对所传入的`io.Reader`，该方法创建了一个带缓冲的`reader`，并将其中的每一行轮流传入解析函数中。通常，对于文本文件，我们会对`io.EOF`进行特殊处理，以便无论它是否以新行结尾其最后一行都能被读取。（当然，对于这种格式，这样做相当自由。）

按照常规，从行号1开始，该文件被逐行读取。第一行用于检查文件是否有个合法的类型和版本号，因此处理实际数据时，行号（`lino`）从2开始读起。

由于我们逐行工作，并且每一个发票文件都表示成两行甚至多行（一行 **INVOICE** 行和一行或者多行**ITEM**行），我们需跟踪当前发票，以便每读一行就可以将其添加到当前发票数据中。这很容易做到，因为所有的发票数据都被追加到一个发票切片中，因此当前发票永远是处于位置`invoices[len(invoices)-1]`处的发票。

当`parseTxtLine()`函数解析一个**INVOICE**行时，它会创建一个新的**Invoice**值，并将一个指向该值的指针追加到`invoices`切片中。

如果要在一个函数内部往一个切片中追加数据，有两种技术可以使用。第一种技术是传入一个指向切片的指针，然后在所指向的切片中操作。第二种技术是传入切片值，同时返回（可能被修改过的）切片给调用者，以赋值回原始切片。`parseTxtLine()`函数使用第二种技术。（我们在前文已看过一个使用第一种技术的例子。）

```

func parseTxtLine(lino int, line string, invoices []*Invoice) ([]*Invoice,
error) {
    var err error
    if strings.HasPrefix(line, "INVOICE") {
        var invoice *Invoice
        invoice, err = parseTxtInvoice(lino, line)
        invoices = append(invoices, invoice)
    } else if strings.HasPrefix(line, "ITEM") {
        if len(invoices) == 0 {
            err = fmt.Errorf("item outside of an invoice line %d", lino)
        } else {
            var item *Item
            item, err = parseTxtItem(lino, line)
            items := &invoices[len(invoices)-1].Items    ①
            *items = append(*items, item)
        }
    }
    return invoices, err
}

```

该函数接受一个行号（`lino`，用于错误报告），需被解析的行，以及我们需要填充的发票切片。

如果该行以文本“INVOICE”开头，我们就调用 `parseTxtInvoice()` 函数来解析该行并创建一个 `Invoice` 值，并返回一个指向它的指针。然后，我们将该 `*Invoice` 值追加到 `invoices` 切片中，并在最后返回该 `invoices` 切片和 `nil` 值或者错误值。需注意的是，这里的发票信息是不完整的，我们只有它的ID、客户 ID、创建和持续时间、是否支付以及注释信息，但是没有任何发票项。

如果该行以“ITEM”开头，我们首先检查当前发票是否存在（即 `invoices` 切片不为空）。如果存在，我们调用 `parseTxtItem()` 函数来解析该行并创建一个 `Item` 值，然后返回一个指向该值的指针。然后我们将该项添加到当前发票的项中。这可以通过取得指向当前发票项的指针（见标注①）以及将指针的值设置为追加新 `*Item` 后的结果来达到。当然，我们本可以使用 `invoices[len(invoices)-1].Items = append(invoices[len(invoices)-1].Items, item)` 来直接添加 `*Item`。

任何其他行（例如空和换页行）都被忽略。顺便提一下，理论上而言，如果我们优先处理“ITEM”的情况该函数会更快，因为数据中发票项的行数远比发票和空行的行数多。

```
func parseTxtInvoice(lino int, line string) (invoice *Invoice, err error) {
    invoice = &Invoice{}
    var raised, due string
    if _, err = fmt.Sscanf(line, "INVOICE ID=%d CUSTOMER=%d" +
        "RAISED=%s      DUE=%s      PAID=%t",    &invoice.Id,
        &invoice.CustomerId,
        &raised, &due, &invoice.Paid); err != nil {
        return nil, fmt.Errorf("invalid invoice %v line %d", err, lino)
    }
    if invoice.Raised, err = time.Parse(dateFormat, raised); err != nil {
        return nil, fmt.Errorf("invalid raised %v line %d", err, lino)
    }
    if invoice.Due, err = time.Parse(dateFormat, due); err != nil {
        return nil, fmt.Errorf("invalid due %v line %d", err, lino)
    }
    if i := strings.Index(line, noteSep); i > -1 {
        invoice.Note = strings.TrimSpace(line[i+len(noteSep):])
    }
}
```

```

    }
    return invoice, nil
}

```

函数开始处，我们创建了一个0值的Invoice值，并将指向它的指针赋值给invoice变量（类型为*Invoice）。扫描函数可以处理字符串、数字以及布尔值，但不能处理time.Time值，因此我们将创建以及持续时间以字符串的形式输入，并单独解析它们。表8-2中列出了扫描函数。

表8-2 fmt中的扫描函数

参数 *r* 是一个从其中读数据的 io.Reader，*s* 是一个从其中读数据的字符串，*fs* 是一个用于 fmt 包中打印函数的格式化字符串（参见表 3-4），*args* 表示一个或者多个需填充的值的指针。所有这些扫描函数返回成功解析（即填充）项的数量，以及另一个空或者非空的错误值。

语法	描述
fmt.Fscan(<i>r</i> , <i>args</i>)	读取 <i>r</i> 中连续的空格或者空行分隔值以填充 <i>args</i>
fmt.Fscanf(<i>r</i> , <i>fs</i> , <i>args</i>)	读取 <i>r</i> 中连续的空格分隔的指定为 <i>fs</i> 格式的值以填充 <i>args</i>

续表

语法	描述
fmt.Fscanln(<i>r</i> , <i>args</i>)	读取 <i>r</i> 中连续的空格分隔的值以填充 <i>args</i> ，同时以新行或者 io.EOF 结尾
fmt.Scan(<i>args</i>)	读取 os.Stdin 中连续的空行分隔的值以填充 <i>args</i>
fmt.Scanf(<i>fs</i> , <i>args</i>)	读取 os.Stdin 中连续的空格分隔的指定为 <i>fs</i> 格式的值以填充 <i>args</i>
fmt.Scanln(<i>args</i>)	读取 os.Stdin 中连续的空格分隔的值以填充 <i>args</i> ，以新行或 io.EOF 结束
fmt.Sscan(<i>s</i> , <i>args</i>)	读取 <i>s</i> 中连续的空行分隔的值以填充 <i>args</i>
fmt.Sscanf(<i>s</i> , <i>fs</i> , <i>args</i>)	读取 <i>s</i> 中连续的空格分隔的指定为 <i>fs</i> 格式的值以填充 <i>args</i>
fmt.Sscanln(<i>s</i> , <i>args</i>)	读取 <i>s</i> 中连续的空格分隔的值以填充 <i>args</i> ，同时以新行或者 io.EOF 结束

如果fmt.Sscanf()函数不能读入与我们所提供的值相同数量的项，或者如果发生了错误（例如，读取错误），函数就会返回一个非空的

错误值。

日期使用`time.Parse()`函数来解析，这在之前的节中已有阐述。如果发票行有冒号，则意味着该行末尾处有注释，那么我们就删除其空白符，并将其返回。我们使用了表达式`line[i+1:]`而非`line[i+len(noteSep):]`，因为我们知道`noteSep`的冒号字符占用了一个UTF-8字节，但为了更为健壮，我们选择了对任何字符都有效的方法，无论它占用多少字节。

```
func parseTxtItem(lino int, line string) (item *Item, err error) {
    item = &Item{}
    if _, err = fmt.Sscanf(line, "ITEM ID=%s PRICE=%f
QUANTITY=%d",
        &item.Id, &item.Price, &item.Quantity); err != nil {
        return nil, fmt.Errorf("invalid item %v line %d", err, lino)
    }
    if i := strings.Index(line, noteSep); i > -1 {
        item.Note = strings.TrimSpace(line[i+len(noteSep):])
    }
    return item, nil
}
```

该函数的功能如我们所见过的`parseTxtInvoice()`函数一样，区别在于除了注释文本之外，所有的发票项值都可以直接扫描。

```
func checkTxtVersion(bufferReader *bufio.Reader) error {
    var version int
    if _, err := fmt.Fscanf(bufferedReader, "INVOICES %d\n",
        &version);
        err != nil {
            return errors.New("cannot read non-invoices text file")
        }
    }
```

```

    } else if version > fileVersion {
        return fmt.Errorf("version %d is too new to read", version)
    }
    return nil
}

```

该函数用于读取发票文本文件的第一行数据。它使用 `fmt.Fscanf()` 函数来直接读取 `bufio.Reader`。如果该文件不是一个发票文件或者其版本太新而不能处理，就会报告错误。否则，返回 `nil` 值。

使用 `fmt` 包的打印函数来写文本文件比较容易。解析文本文件却挑战不小，但是 Go 语言的 `regexp` 包中提供了 `strings.Fields()` 和 `strings.Split()` 函数，`fmt` 包中提供了扫描函数，使得我们可以很好的解决该问题。

8.1.4 处理Go语言二进制文件

Go 语言的二进制（`gob`）格式是一个自描述的二进制序列。从其内部表示来看，Go 语言的二进制格式由一个 0 块或者更多块的序列组成，其中的每一块都包含一个字节数，一个由 0 个或者多个 `typeId-typeSpecification` 对组成的序列，以及一个 `typeId-value` 对。如果 `typeId-value` 对的 `typeId` 是预先定义好的（例如，`bool`、`int` 和 `string` 等），则这些 `typeId-typeSpecification` 对可以省略。否则就用类型对来描述一个自定义类型（如一个自定义的结构体）。类型对和值对之间的 `typeId` 没有区别。正如我们将看到的，我们无需了解其内部结构就可以使用 `gob` 格式，因为 `encoding/gob` 包会在幕后为我们打理好一切底层细节 [2]。

`encoding/gob` 包也提供了与 `encoding/json` 包一样的编码解码功能，并且容易使用。通常而言，如果对肉眼可读性不做要求，`gob` 格式是 Go 语言上用于文件存储和网络传输最为方便的格式。

8.1.4.1 写Go语言二进制文件

下面有个方法用于将整个[]*Invoice项的数据以gob的格式写入一个打开的文件（或者是任何满足io.Writer接口的值）中。

```
type GobMarshaler struct{}

func (GobMarshaler) MarshalInvoices(writer io.Writer, invoices
[]*Invoice) error {
    encoder := gob.NewEncoder(writer)
    if err := encoder.Encode(magicNumber); err != nil {
        return err
    }
    if err := encoder.Encode(fileVersion); err != nil {
        return err
    }
    return encoder.Encode(invoices)
}
```

在方法开始处，我们创建了一个包装了io.Writer的gob编码器，它本身是一个writer，让我们可以写数据。

我们使用gob.Encoder.Encode()方法来写数据。该方法能够完美地处理我们的发票切片，其中每个发票切片包含它自身的发票项切片。该方法返回一个空或者非空的错误值。如果发生错误，则立即返回给它的调用者。

往文件写入幻数（magic number）和文件版本并不是必需的，但正如将在练习中所看到的那样，这样做可以在后期更方便地改变文件格式。

需注意的是，该方法并不真正关心它编码数据的类型，因此创建类似的函数来写gob数据区别不大。此外，GobMarshaler.MarshalInvoices()方法无需任何改变就可以写新数据格式。

由于Invoice结构体的字段都是布尔值、数字、字符串、time.Time值以及包含布尔值、数字、字符串和time.Time值的结构体（如Item），这里的代码可以正常工作。

如果我们的结构体包含某些不可用gob格式编码的字段，那么就必須更改该结构体以便满足gob.GobEncoder和gob.GobDecoder接口。该gob编码器足够智能来检查它需要编码的值是不是一个gob.GobEncoder，如果是，那么编码器就使用该值自身的GobEncode()方法而非编码器内置的编码方法来编码。相同的规则也作用于解码时，检查该值是否定义了GobDecode()方法以满足gob.GobDecoder接口。（该invoicedata例子的源代码gob.go文件中包含了相应的代码，将Invoice定义成一个编码器和解码器。因为这些代码不是必須的，因此我们将其注释掉，只是为了演示如何做。）让一个结构体满足这些接口会极大地降低gob的读写速度，也会产生更大的文件。

8.1.4.2 读Go语言二进制文件

读gob数据和写一样简单，如果我们的目标数据类型与写时相同。GobMarshaler.UnmarshalInvoices()方法接受一个io.Reader（例如，一个打开的文件），并从中读取gob数据。

```
func (GobMarshaler) UnmarshalInvoices(reader io.Reader)([*Invoice,
error) {
    decoder := gob.NewDecoder(reader)
    var magic int
    if err := decoder.Decode(&magic); err != nil {
        return nil, err
    }
    if magic != magicNumber {
        return nil, errors.New("cannot read non-invoices gob file")
    }
}
```

```

var version int
if err := decoder.Decode(&version); err != nil {
    return nil, err
}
if version > fileVersion {
    return nil, fmt.Errorf("version %d is too new to read", version)
}
var invoices []*Invoice
err := decoder.Decode(&invoices)
return invoices, err
}

```

我们有3项数据要读：幻数、文件版本号以及所有发票数据。`gob.Decoder.Decode()`方法接受一个指向目标值的指针，返回一个空或者非空的错误值。我们使用头两个变量（幻数和版本号）来确认我们得到的是一个gob格式的发票文件，并且该文件的版本是我们可以处理的。然后，我们读取发票文件，在此过程中，`gob.Decoder.Decode()`方法会根据所读取的发票数据增加`invoices`切片的大小，并根据需要来将指向函数实时创建的`Invoices`数据（及其发票项）的指针保存在`invoices`切片中。最后，该方法返回`invoices`切片，以及一个空的错误值，或者如果发生问题则返回非空的错误值。

如果发票数据由于添加了导出字段被更改了，针对布尔值、整数、字符串、`time.Time`值以及包含这些类型值的结构体，该方法还能继续工作。当然，如果数据包含其他类型，那就必须更新方法以满足`gob.GobEncoder`和`gob.GobDecoder`接口。

处理结构体类型时，`gob`格式非常灵活，能够无缝地处理一些不同的数据结构。例如，如果一个包含某值的结构体被写成gob格式，那么就必然可以从gob格式中将该值读回到此结构体，甚至也读回到许多其

他类似的结构体，比如包含指向该值指针的结构体，或者结构体中的值类型兼容也可（比如`int`相对于`uint`，或者类似的情况）。同时，正如`invoicedata`示例所示，`gob`格式可以处理嵌套的数据（但是，在本书撰写时，它还不能处理递归的值）。`gob`的文档中给出了它能处理的格式以及该格式的底层存储结构，但如果我们使用相同的类型来进行读写，正如上例中所做的那样，我们就不必关心这些。

8.1.5 处理自定义的二进制文件

虽然Go语言的`encoding/gob`包非常易用，而且使用时所需代码量也非常少，我们仍有可能需要创建自定义的二进制格式。自定义的二进制格式有可能做到最紧凑的数据表示，并且读写速度可以非常快。不过，在实际使用中，我们发现以Go语言二进制格式的读写通常比自定义格式要快非常多，而且创建的文件也不会大很多。但如果我们必须通过满足`gob.GobEncoder`和`gob.GobDecoder`接口来处理一些不可被`gob`编码的数据，这些优势就有可能失去。在有些情况下我们可能需要与一些使用自定义二进制格式的软件交互，因此了解如何处理二进制文件就非常有用。

图8-1给出了`.inv`自定义二进制格式如何表示一个发票文件的概要。整数值表示成固定大小的无符号整数。布尔值中的`true`表示成一个`int8`类型的值1，`false`表示成0。字符串表示成一个字节数（类型为`int32`）后跟一个它们的UTF-8编码的字节切片`[]byte`。对于日期，我们采取稍微非常规的做法，将一个ISO-8601格式的日期（不含连字符）当成一个数字，并将其表示成`int32`值。例如，我们将日期2006-01-02表示成数字20 060 102。每一个发票项表示成一个发票项的总数后跟各个发票项。（回想一下，发票项ID是字符串而非整数，这与发票ID不同，参见8.1节。）

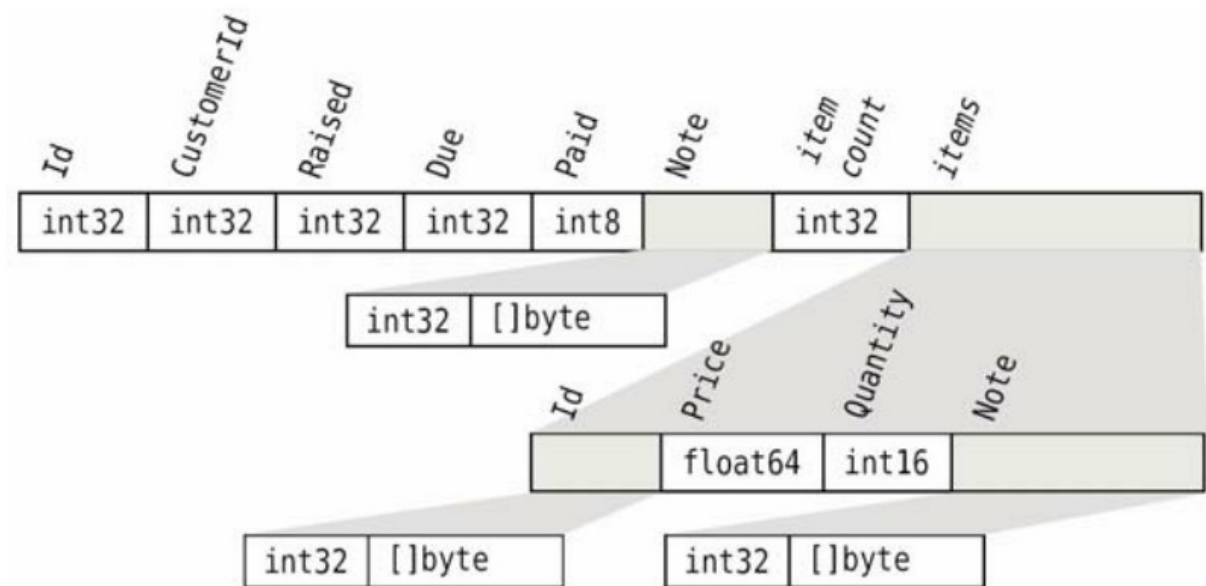


图8-1.inv自定义二进制格式

8.1.5.1 写自定义二进制文件

encoding/binary包中的binary.Write()函数使得以二进制格式写数据非常简单。

```
type InvMarshaler struct{}
var byteOrder = binary.LittleEndian
func (InvMarshaler) MarshalInvoices(writer io.Writer, invoices
[]*Invoice) error {
    var write invWriterFunc = func(x interface{}) error {
        return binary.Write(writer, byteOrder, x)
    }
    if err := write(uint32(magicNumber)); err != nil {
        return err
    }
    if err := write(uint16(fileVersion)); err != nil {
        return err
    }
}
```

```

    if err := write(int32(len(invoices))); err != nil {
        return err
    }
    for _, invoice := range invoices {
        if err := write.writeInvoice(invoice); err != nil {
            return err
        }
    }
    return nil
}

```

该方法将所有发票项写入给定的`io.Writer`中。它开始时创建了一个便捷的`write()`函数，该函数能够捕获我们要使用的`io.Writer`和字节序。正如处理`.txt`格式所做的那样，我们将`write()`函数定义为一个特定的类型（`invWriterFunc`），并且为该`write()`函数创建了一些方法（例如`invWriterFunc.WriteInvoices()`），以便后续使用。

需注意的是，读和写二进制数据时其字节序必须一致。（我们不能将`byteOrder`定义为一个常量，因为`binary.LittleEndian`或者`binary.BigEndian`不是像字符串或者整数这样的简单值。）

这里，写数据的方式与我们之前在看到写其他格式数据的方式类似。一个非常重要的不同在于，将幻数和文件版本写入后，我们写入了一个表示发票数量的数字。（也可以跳过而不写该数字，而只是简单地将发票写入。然后，读数据的时候，持续地依次读入发票直到遇到`io.EOF`。）

```

type invWriterFunc func(interface{}) error
func (write invWriterFunc) writeInvoice(invoice *Invoice) error {
    for _, i := range []int{invoice.Id, invoice.CustomerId} {
        if err := write(int32(i)); err != nil {

```



```

        return err
    }
}
for _, date := range []time.Time{invoice.Raised, invoice.Due} {
    if err := write.writeDate(date); err != nil {
        return err
    }
}
if err := write.writeBool(invoice.Paid); err != nil {
    return err
}
if err := write.writeString(invoice.Note); err != nil {
    return err
}
if err := write.writeInt32(len(invoice.Items)); err != nil {
    return err
}
for _, item := range invoice.Items {
    if err := write.writeItem(item); err != nil {
        return err
    }
}
return nil
}

```

对于每一个发票数据，`writeInvoice()`方法都会被调用一遍。它接受一个指向被写发票数据的指针，并使用作为接收器的`write()`函数来写数据。

该方法开始处以int32写入了发票ID及客户ID。当然，以纯int型写入数据是合法的，但底层机器以及所使用的Go语言版本的改变都可能导致int的大小改变，因此写入时非常重要的一点是确定整型的符号和大小，如 uint32和int32 等。接下来，我们使用自定义的writeDate()方法写入创建和过期时间，然后写入表示是否支付的布尔值和注释字符串。最后，我们写入了一个代表发票中有多少发票项的数字，随后再使用writeItem()方法写入发票项。

```
const invDateFormat = "20060102" // 必须总是使用该日期值
func (write invWriterFunc) writeDate(date time.Time) error {
    i, err := strconv.Atoi(date.Format(invDateFormat))
    if err != nil {
        return err
    }
    return write(int32(i))
}
```

前文中我们讨论了time.Time.Format()函数以及为何必须在格式字符串中使用特定的日期2006-01-02。这里，我们使用了类ISO-8601格式，并去除连字符以便得到一个八个数字的字符串，其中如果月份和天数为单一数字则在其前面加上0。然后，将该字符串转换成数字。例如，如果日期是2012-08-05，则将其转换成一个等价的数字，即20120805，然后以int32的形式将该数字写入。

值得一提的是，如果我们想存储日期/时间值而非仅仅是日期值，或者只想得到一个更快的计算，我们可以将该方法的调用替换成调用write(int64(date.Unix()))，以存储一个Unix新纪元以来的秒数。相应的读取数据的方法则类似于var d int64;if err:=binary.Read(reader, byteOrder, &d); err != nil { return err }; date := time.Unix(d, 0)。

```
func (write invWriterFunc) writeBool(b bool) error {
```

```

var v int8
if b {
    v = 1
}
return write(v)
}

```

本书撰写时，`encoding/binary`包还不支持读写布尔值，因此我们创建了该简单方法来处理它们。顺便提一下，我们不必使用类型转换（如`int8(v)`），因为变量`v`已经是一个有符号并且固定大小的类型了。

```

func (write invWriterFunc) writeString(s string) error {
    if err := write(int32(len(s))); err != nil {
        return err
    }
    return write([]byte(s))
}

```

字符串必须以它们底层的UTF-8编码字节的形式来写入。这里，我们首先写入了所需写入的字节总数，然后再写入所有字节。（如果数据是固定宽度的，就不需要写入字节数。当然，前提是，读取数据时，我们创建了一个存储与写入的数据大小相同的空切片`[]byte`。）

```

func (write invWriterFunc) writeItem(item *Item) error {
    if err := write.writeString(item.Id); err != nil {
        return err
    }
    if err := write(item.Price); err != nil {
        return err
    }
    if err := write(int16(item.Quantity)); err != nil {

```

```

        return err
    }
    return write.writeString(item.Note)
}

```

该方法用于写入一个发票项。对于字符串 ID 和注释文本，我们使用 `invWriterFunc.writeString()` 方法，对于物品数量，我们使用无符号的大小固定的整数。但是对于价格，我们就以它原始的形式写入，因为它本来就是固定大小的类型（`float64`）。

往文件中写入二进制数据并不难，只要我们小心地将可变长度数据的大小在数据本身前面写入，以便读数据时知道该读多少。当然，使用 `gob` 格式非常方便，但是使用一个自定义的二进制格式所产生的文件更小。

8.1.5.2 读自定义二进制文件

读取自定义的二进制数据与写自定义二进制数据一样简单。我们无需解析这类数据，只需使用与写数据时相同的字节顺序将数据读进相同类型的值中。

```

func (InvMarshaler) UnmarshalInvoices(reader io.Reader) ([]*Invoice,
error){
    if err := checkInvVersion(reader); err != nil {
        return nil, err
    }
    count, err := readIntFromInt32(reader)
    if err != nil {
        return nil, err
    }
    invoices := make([]*Invoice, 0, count)
    for i := 0; i < count; i++ {

```

```

        invoice, err := readInvInvoice(reader)
        if err != nil {
            return nil, err
        }
        invoices = append(invoices, invoice)
    }
    return invoices, nil
}

```

该方法首先检查所给定版本的发票文件能否被处理，然后使用自定义的`readIntFromInt32()`函数从文件中读取所需处理的发票数量。我们将 `invoices` 切片的长度设为 0（即当前还没有发票），但其容量正好是我们所需要的。然后，轮流读取每一个发票并将其存储在`invoices`切片中。

另一种可选的方法是使用`make([]*Invoice, count)`代替`make()`，使用 `invoices[i] = invoice` 代替 `append()`。不管怎样，我们倾向于使用所需的容量来创建切片，因为与实时增长切片相比，这样做更有潜在的性能优势。毕竟，如果我们再往一个其长度与容量相等的切片中追加数据，切片会在背后创建一个新的容量更大的切片，并将起原始切片数据复制至新切片中。然而，如果其容量一开始就足够，后面就没必要进行复制。

```

func checkInvVersion(reader io.Reader) error {
    var magic uint32
    if err := binary.Read(reader, byteOrder, &magic); err != nil {
        return err
    }
    if magic != magicNumber {
        return errors.New("cannot read non-invoices inv file")
    }
}

```

```

    }
    var version uint16
    if err := binary.Read(reader, byteOrder, &version); err != nil {
        return err
    }
    if version > fileVersion {
        return fmt.Errorf("version %d is too new to read", version)
    }
    return nil
}

```

该函数试图从文件中读取其幻数及版本号。如果该文件格式可接受，则返回`nil`；否则返回非空错误值。

其中的`binary.Read()`函数与 `binary.Write()`函数相对应，它接受一个从中读取数据的`io.Reader`、一个字节序以及一个指向特定类型的用于保存所读数据的指针。

```

func readIntFromInt32(reader io.Reader) (int, error) {
    var i32 int32
    err := binary.Read(reader, byteOrder, &i32)
    return int(i32), err
}

```

该辅助函数用于从二进制文件中读取一个`int32`值，并以`int`类型返回。

```

func readInvInvoice(reader io.Reader) (invoice *Invoice, err error) {
    invoice = &Invoice{}
    for _, pId := range []*int{&invoice.Id, &invoice.CustomerId} {
        if *pId, err = readIntFromInt32(reader); err != nil {
            return nil, err
        }
    }
}

```

```

    }
}
for _, pDate := range []*time.Time{&invoice.Raised, &invoice.Due}
{
    if *pDate, err = readInvDate(reader); err != nil {
        return nil, err
    }
}
if invoice.Paid, err = readBoolFromInt8(reader); err != nil {
    return nil, err
}
if invoice.Note, err = readInvString(reader); err != nil {
    return nil, err
}
var count int
if count, err = readIntFromInt32(reader); err != nil {
    return nil, err
}
invoice.Items, err = readInvItems(reader, count)
return invoice, err
}

```

每次读取发票文件的时候，该函数都会被调用。函数开始处创建了一个初始化为零值的**Invoice**值，并将指向它的指针保存在**invoice**变量中。

发票**ID**和客户**ID**使用自定义的**readIntFromInt32()**函数读取。这段代码的微妙之处在于，我们迭代那些指向发票**ID**和客户**ID**的指针，并将返回的整数赋值给指针（**pId**）所指的值。

一个可选的方案是单独处理每一个 ID。例如，`if invoice.Id, err = readIntFromInt32(reader); err != nil { return err}`等。

读取创建及过期日期的流程与读取 ID 的流程完全一样，只是这次我们使用的是自定义的`readInvDate()`函数。

正如读取ID一样，我们也可以以更加简单的方式单独处理日期。例如，`if invoice.Due, err = readInvDate(reader); err != nil { return err}`等。

稍后将看到，我们使用一些辅助函数读取是否支付的标志和注释文本。发票数据读完之后，我们再读取有多少个发票项，然后调用`readInvItems()`函数读取全部发票项，传递给该函数一个用于读取的`io.Reader`值和一个表示需要读多少项的数字。

```
func readInvDate(reader io.Reader) (time.Time, error) {  
    var n int32  
    if err := binary.Read(reader, byteOrder, &n); err != nil {  
        return time.Time{}, err  
    }  
    return time.Parse(invDateFormat, fmt.Sprint(n))  
}
```

该函数用于读取表示日期的`int32`值（如20130501），并将该数字解析成字符串表示的日期值，然后返回对应的`time.Time`值（如2013-05-01）。

```
func readBoolFromInt8(reader io.Reader) (bool, error) {  
    var i8 int8  
    err := binary.Read(reader, byteOrder, &i8)  
    return i8 == 1, err  
}
```

该简单的辅助函数读取一个`int8`数字，如果该数字为1则返回`true`，否则返回`false`。


```

func readInvString(reader io.Reader) (string, error) {
    var length int32
    if err := binary.Read(reader, byteOrder, &length); err != nil {
        return "", nil
    }
    raw := make([]byte, length)
    if err := binary.Read(reader, byteOrder, &raw); err != nil {
        return "", err
    }
    return string(raw), nil
}

```

该函数读取一个[]byte 切片，但它的原理适用于任何类型的切片，只要写入切片之前写明了切片中包含多少项元素。

函数首先将切片项的个数读到一个length变量中。然后创建一个长度与此相同的切片。给binary.Read()函数传入一个指向切片的指针之后，它就会往该切片中尽可能地读入该类型的项（如果失败则返回一个非空的错误值）。需注意的是，这里重要的是切片的长度，而非其容量（其容量可能等于或者大于长度）。

在本例中，该[]byte切片保存了UTF-8编码的字节，我们将其转换成字符串后将其返回。

```

func readInvItems(reader io.Reader, count int) ([]*Item, error) {
    items := make([]*Item, 0, count)
    for i := 0; i < count; i++ {
        item, err := readInvItem(reader)
        if err != nil {
            return nil, err
        }
    }
}

```

```

        items = append(items, item)
    }
    return items, nil
}

```

该函数读入发票的所有发票项。由于传入了一个计数值，因此它知道应该读入多少项。

```

func readInvItem(reader io.Reader) (item *Item, err error) {
    item = &Item{}
    if item.Id, err = readInvString(reader); err != nil {
        return nil, err
    }
    if err = binary.Read(reader, byteOrder, &item.Price); err != nil {
        return nil, err
    }
    if item.Quantity, err = readIntFromInt16(reader); err != nil {
        return nil, err
    }
    item.Note, err = readInvString(reader)
    return item, nil
}

```

该函数读取单个发票项。从结构上看，它与 `readInvInvoice()` 函数类似，首先创建一个初始化为零值的 `Item` 值，并将指向它的指针存储在变量 `item` 中，然后填充该 `item` 变量的字段。价格可以直接读入，因为它是以 `float64` 类型写入文件的，是一个固定大小的类型。`Item.Price` 字段的类型也一样。（我们省略了 `readIntFromInt16()` 函数，因为它与我们前文所描述的 `readIntFromInt32()` 函数基本相同。）

至此，我们完成了对自定义二进制数据的读和写。只要小心选择表示长度的整数符号和大小，并将该长度值写在变长值（如切片）的内容之前，那么使用二进制数据进行工作并不难。

Go语言对二进制文件的支持还包括随机访问。这种情况下，我们必须使用`os.OpenFile()`函数来打开文件（而非`os.Open()`），并给它传入合理的权限标志和模式（例如，`os.O_RDWR`表示可读写）参数 [3]。然后，就可以使用`os.File.Seek()`方法来在文件中定位并读写，或者使用`os.File.ReadAt()`和`os.File.WriteAt()`方法来从特定的字节偏移中读取或者写入数据。Go语言还提供了其他常用的方法，包括`os.File.Stat()`方法，它返回的`os.FileInfo`包含了文件大小、权限以及日期时间等细节信息。

8.2 归档文件

Go语言的标准库提供了对几种压缩格式的支持，其中包括gzip，因此Go程序可以无缝地读写.gz扩展名的gzip压缩文件或非.gz扩展名的非压缩文件。此外，标准库也提供了读和写.zip文件、tar包文件（.tar和.tar.gz），以及读.bz2文件（即.tar.bz2文件）的功能。

本节中我们会看一些从两个程序中抽出的代码。第一个是pack程序（在文件pack/pack.go中），它从命令行接受一个归档文件的文件名和需打包的文件列表。它通过检测归档文件的扩展名来判断该使用何种打包格式。第二个是unpack程序（在文件unpack/unpack.go中），也从命令行接受一个归档文件的文件名，并从中提取所有打包的文件，如有必要则在提取过程中重建目录结构。

8.2.1 创建zip归档文件

要使用 `zip` 包来压缩文件，我们首先必须打开一个用于写的文件，然后创建一个`*zip.Writer`值来往其中写入数据。然后，对于每一个我们希望加入`.zip`归档文件的文件，我们必须读取该文件并将其内容写入`*zip.Writer`中。该`pack`程序使用了`createZip()`和`writeFileToZip()`两个函数以这种方式来创建一个`.zip`文件。

```
func createZip(filename string, files []string) error {
    file, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer file.Close()
    zipper := zip.NewWriter(file)
    defer zipper.Close()
    for _, name := range files {
        if err := writeFileToZip(zipper, name); err != nil {
            return err
        }
    }
    return nil
}
```

该 `createZip()`函数和`writeFileToZip()`函数都比较简短，因此容易让人觉得应该写入一个函数中。这是不明智的，因为在该 `for` 循环中我们可能打开一个又一个的文件（即`files` 切片中的所有文件），从而可能超出操作系统允许的文件打开数上限。这点我们在前面章节中已有简短的阐述。当然，我们可以在每次迭代中调用`os.File.Close()`，而非延迟执行它，但这样做还必须保证程序无论是否出错文件都必须关闭。

因此，最为简便而干净的解决方案是，像这里所做的那样，总是创建一个独立的函数来处理每个独立的文件。

```
func writeFileToZip(zipper *zip.Writer, filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close()
    info, err := file.Stat()
    if err != nil {
        return err
    }
    header, err := zip.FileInfoHeader(info)
    if err != nil {
        return err
    }
    header.name = sanitizedName(filename)
    writer, err := zipper.CreateHeader(header)
    if err != nil {
        return err
    }
    _, err = io.Copy(writer, file)
    return err
}
```

首先我们打开需要归档的文件以供读取，然后延迟关闭它。这是我们处理文件的老套路了。

接下来，我们调用`os.File.Stat()`方法来取得包含时间戳和权限标志的`os.FileInfo`值。然后，我们将该值传给`zip.FileInfoHeader()`函数，该函数返回一个`zip.FileHeader`值，其中保存了时间戳、权限以及文件名。在压缩文件中，我们无需使用与原始文件名一样的文件名，因此这里我们使用净化过的文件名来覆盖原始文件名（保存在`zip.FileHeader.Name`字段中）。

头部设置好之后，我们将其作为参数调用`zip.CreateHeader()`函数。这会在`.zip`压缩文件中创建一个项，其中包含头部的时间戳、权限以及文件名，并返回一个`io.Writer`，我们可以往其中写入需要被压缩的文件的内容。为此，我们使用了`io.Copy()`函数，它能够返回所复制的字节数（我们已将其丢弃），以及一个为空或者非空的错误值。

如果在任何时候发生错误，该函数就会立即返回并由调用者处理错误。如果最终没有错误发生，那么该`.zip`压缩文件就会包含该给定文件。

```
func sanitizedName(filename string) string{
    if len(filename) > 1 && filename[1] == ':' &&
        runtime.GOOS == "windows" {
        filename = filename[2:]
    }
    filename = filepath.ToSlash(filename)
    filename = strings.TrimLeft(filename, ".")
    return strings.Replace(filename, "../", "", -1)
}
```

如果一个归档文件中包含的文件带有绝对路径或者含有“..”路径组件，我们就有可能在解开归档的时候意外覆盖本地重要文件。为了降低这种风险，我们对保存在归档文件里每个文件的文件名都做了相应的净化。

该 `sanitizedName()` 函数会删除路径头部的盘符以及冒号（如果有的话），然后删除头部任何目录分隔符、点号以及任何“..”路径组件，并将文件分隔符强制转换成正向斜线。

8.2.2 创建可压缩的tar包

创建tar归档文件与创建.zip归档文件非常类似，主要不同点在于我们将所有数据都写入相同的writer中，并且在写入文件的数据之前必须写入完整的头部，而非仅仅是一个文件名。我们在该pack程序的实现中使用了`createTar()`和`writeFileToTar()`函数。

```
func createTar(filename string, files []string) error {
    file, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer file.Close()
    var fileWriter io.WriterCloser = file
    if strings.HasSuffix(filename, ".gz") {
        fileWriter = gzip.NewWriter(file)
        defer fileWriter.Close()
    }
    writer := tar.NewWriter(fileWriter)
    defer writer.Close()
    for _, name := range files {
        if err := writeFileToTar(writer, name); err != nil {
            return err
        }
    }
}
```

```

    }
    return nil
}

```

该函数创建了包文件，而且如果扩展名显示该 tar 包需要被压缩则添加一个 gzip 过滤。gzip.NewWriter()函数返回一个*gzip.Writer值，它满足io.WriteCloser接口（正如打开的*os.File一样）。

一旦文件准备好写入，我们创建一个*tar.Writer 往其中写入数据。然后迭代所有文件并将每一个写入归档文件。

```

func writeFileToTar(writer *tar.Writer, filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close()
    stat, err := file.Stat()
    if err != nil {
        return err
    }
    header := &tar.Header{
        Name:      sanitizedName(filename),
        Mode:      int64(stat.Mode()),
        Uid:       os.Getuid(),
        Gid:       os.Getuid(),
        Size:      stat.Size(),
        ModTime:   stat.ModTime(),
    }
    if err = writer.WriteHeader(header); err != nil {

```



```

        return err
    }
    _, err = io.Copy(writer, file)
    return err
}

```

函数首先打开需要处理的文件并设置延迟关闭。然后调用Stat()方法取得文件的模式、大小以及修改日期/时间。这些信息用于填充*tar.Header，每个文件都必须创建一个tar.Header结构并写入到tar归档文件里，（此外，我们设置了头部的用户以及组ID，这会在类Unix系统中用到。）我们必须至少设置头部的文件名（其Name字段）以及表示文件大小的Size字段，否则这个.tar包就是非法的。

当*tar.Header结构体创建好后，我们将它写入归档文件，再接着写入文件的内容。

8.2.3 解开zip归档文件

解开一个.zip归档文件与创建一个归档文件一样简单，只是如果归档文件中包含带有路径的文件名，就必须重建目录结构。

```

func unpackZip(filename string) error {
    reader, err := zip.OpenReader(filename)
    if err != nil {
        return err
    }
    defer reader.Close()
    for _, zipFile := range reader.Reader.File {
        name := sanitized_name(zipFile.Name)
        mode := zipFile.Mode()
    }
}

```

```

        if mode.IsDir() {
            if err = os.MkdirAll(name, 0755); err != nil {
                return err
            }
        } else {
            if err = unpackZippedFile(name, zipFile); err != nil {
                return err
            }
        }
    }
    return nil
}

```

该函数打开给定的.zip文件用于读取。这里没有使用os.Open()函数来打开文件后调用zip.NewReader()，而是使用zip包提供的zip.OpenReader()函数，它可以方便地打开并返回一个*zip.ReadCloser值让我们使用。zip.ReadCloser最为重要的一点是它包含了导出的zip.Reader结构体字段，其中包含一个包含指向zip.File结构体指针的[]*zip.File切片，其中的每一项表示.zip压缩文件中的一个文件。

我们迭代访问该reader的zip.File结构体，并创建一个净化过的文件及目录名（使用我们在pack程序中用到的sanitizedName()函数），以降低覆盖重要文件的风险。

如果遇到一个目录（由*zip.File的os.FileMode的IsDir()方法报告），我们就创建一个目录。os.MkdirAll()函数传入了有用的属性信息，会自动创建必要的中间目录以创建特定的目标目录，如果目录已经存在则会安全地返回nil而不执行任何操作。[\[4\]](#)如果遇到的是一个文件，则交由自定义的unpackZippedFile()函数进行解压。

```

func unpackZippedFile(filename string, zipFile *zipFile) error {

```

```

writer, err := os.Create(filename)
if err != nil {
return err
}
defer writer.Close()
reader, err := zipFile.Open()
if err != nil {
return err
}
defer reader.Close()
if _, err = io.Copy(writer, reader); err != nil {
return err
}
if filename == zipFile.Name {
fm.Println(filename)
} else {
fmt.Printf("%s [%s]\n", filename, zipFile.Name)
}
return nil
}

```

`unpackZippedFile()`函数的作用就是将`.zip` 归档文件里的单个文件抽取出来，写到`filename`指定的文件里去。首先它创建所需要的文件，然后，使用`zip.File.Open()`函数打开指定的归档文件，并将数据复制到新创建的文件里去。

最后，如果没有错误发生，该函数会往终端打印所创建文件的文件名，如果处理后的文件名与原始文件名不一样，则将原始文件名包含在方括号中。

值得注意的是，该 *zip.File 类型也有一些其他的方法，如 zip.File.Mode()（在前面的 unpackZip() 函数中已有使用），zip.File.ModTime()（以 time.Time 值返回文件的修改时间）以及返回文件的 os.FileInfo 值的 zip.FileInfo()。

8.2.4 解开tar归档文件

解开tar归档文件比创建tar归档文档稍微简单些。然而，跟解开.zip文件一样，如果归档文件中的某些文件名包含路径，必须重建目录结构。

```
func unpackTar(filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close()
    var fileReader io.ReadCloser = file
    if strings.HasSuffix(filename, ".gz") {
        if fileReader, err = gzip.NewReader(file); err != nil {
            return err
        }
        defer fileReader.Close()
    }
    reader := tar.NewReader(fileReader)
    return unpackTarFiles(reader)
}
```

该方法首先按照Go语言的常规方式打开归档文件，并延迟关闭它。如果该文件使用了gzip压缩则创建一个gzip解压缩过滤器并延迟关闭它。gzip.NewReader()函数返回一个*gzip.Reader值，正如打开一个常规文件（类型为*os.File）一样，它也满足io.ReadCloser接口。

设置好了文件reader之后，我们创建一个*tar.Reader来从中读取数据，并将接下来的工作交给一个辅助函数。

```
func unpackTarFiles(reader *tar.Reader) error {
    for {
        header, err := reader.Next()
        if err != nil {
            if err == io.EOF {
                return nil // OK
            }
            return err
        }
        filename := sanitizedHeaderName(header.Name)
        switch header.Typeflag {
        case tar.TypeDir:
            if err = os.MkdirAll(filename, 0755); err != nil {
                return err
            }
        case tar.TypeReg:
            if err = unpackTarFile(filename, header.Name, reader);
                err != nil {
                return err
            }
        }
    }
}
```

```

    }
    return nil
}

```

该函数使用一个无限循环来迭代读取归档文档中的每一项，直到遇到 `io.EOF`（或者直到遇到错误）。`tar.Next()` 方法返回压缩文档中第一项或者下一项的 `*tar.Header` 结构体，失败则报告错误。如果错误值为 `io.EOF`，则意味着读取文件结束，因此返回一个空的错误值。

得到 `*tar.Header` 后，根据该头部的 `Name` 字段创建一个净化过的文件名。然后，通过该项的类型标记来进行 `switch`。对于该简单示例，我们只考虑目录和常规文件，但实际上，归档文件中还可以包含许多其他类型的项（如符号链接）。

如果该项是一个目录，我们按照处理 `.zip` 文件时所采用的方法创建该目录。而如果该项是一个常规文件，我们将其工作交给另一个辅助函数。

```

func unpackTarFile(filename, tarFilename string, reader *tar.Reader)
error{
    writer, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer writer.Close()
    if _, err := io.Copy(writer, reader); err != nil {
        return err
    }
    if filename == tarFilename {
        fmt.Println(filename)
    } else {

```

```

        fmt.Printf("%s [%s]\n", filename, tarFilename)
    }
    return nil
}

```

针对归档文件中的每一项，该函数创建了一个新文件，并延迟关闭它。然后，它将归档文件的下一项数据复制到该文件中。同时，正如在`unpackZippedFile()`函数中所做的那样，我们将刚创建文件的文件名打印到终端，如果净化过的文件名与原始文件名不同，则在方括号中给出原始文件名。

至此，我们完成了对压缩和归档文件及常规文件处理的介绍。Go语言使用 `io.Reader`、`io.ReadCloser`、`io.Writer`和`io.WriteCloser`等接口处理文件的方式让开发者可以使用相同的编码模式来读写文件或者其他流（如网络流或者甚至是字符串），从而大大降低了难度。

8.3 练习

本章有3个练习。第一个练习需要对本章给出的一个示例程序进行简单的修改。第二个练习需要读者从头写一个短小但有难度的程序。第三个练习需要读者对本章给出的一个示例程序做大量修改。

（1）将`unpack`目录拷进一个新的目录，如`my_unpack`，修改`unpack.go`程序，使得它还能够解压缩`.tar.bz2`（`bzip2`压缩的）文件。这需要对一些文件进行小修改，往`unpackTar()`函数中添加的代码大概10行。该更改有一定的挑战，因为`bzip2.NewReader()`函数不返回`io.ReadCloser`。文件`unpack_ans/unpack.go`中给出了一个解决方案，它比原始示例大概多10行代码。

(2) Windows文本文件（.txt）通常使用UTF-16-LE（UTF-16 little-endian）编码。UTF-16编码的文件必须以一个字节序标记开头，[0xFF, 0xFE]表示以低字节序存储，[0xFE, 0xFF]表示以高字节序存储。编写一个程序，它能够从命令行读取一个 UTF-16 编码的文件，并将其文本以UTF-8编码的形式写入os.Stdout或者一个从命令行输入的文件中。请确保按正确的方式读取以低字节序和高字节序编码的文件。本书示例中提供了一些小的测试文件：utf16-to-utf8/utf-16-be.txt和utf16-to-utf8/utf-16-le.txt。binary包的Read()函数可以使用特定的字节顺序读取 uint16 值（UTF-16 编码的字符就是这类值）。而unicode/utf16包中的Decode()函数可以将一个 uint16 值的切片转换成一个码点切片（即转换成一个[]rune）。因此，使用string()来包装utf16.Decode()调用的结果就足以产生UTF-8编码的字符串。文件utf16-to-utf8/utf16-to-utf8.go文件中给出了一个解决方案，除了导入包的代码之外大概有50行代码。

(3) 将invoicedata目录复制成另一个新目录，如my_invoicedata，根据以下几种方式修改该invoicedata程序。首先，将Invoice和Item结构体改成如下所示结构。

<pre>type Invoice struct { // fileVersion Id int // 100 CustomerId int // 100 DepartmentId string // 101 Raised time.Time // 100 Due time.Time // 100 Paid bool // 100 Note string // 100 Items []*Item // 100 }</pre>	<pre>type Item struct { // fileVersion Id string // 100 Price float64 // 100 Quantity int // 100 TaxBand int // 101 Note string // 100 }</pre>
--	---

现在修改原程序，使得它能够从原始格式读取发票数据，并总是以新的格式（即对应新结构体的格式）写入。

当从原始格式读取数据的时候，不能使用额外字段中的0值，因此需要按如下规则使用相应的值填充这些字段：如果发票的ID小于3000

则将发票的部门ID设为“GEN”，否则如果该ID小于4000则将其设为“MKT”，否则如果该ID小于5000则将其设为“COM”，否则如果该ID小于6000则将其设为“EXP”，否则如果该ID小于7000则将其设为“INP”，否则如果该ID小于8000则将其设为“TZZ”，否则如果该ID小于9000则将其设为“V20”，否则将其设为“X15”。将每一项的税阶设为该项ID第三个字符对应的整数值。例如，如果该ID为“JU4661”，则将其税阶设为4。

目录 `invoicedata_ans` 中提供了一个解决方案。该解决方案往 `invoicedata.go` 文件中添加了3个函数：一个用于更新[]*Invoice切片中的所有发票数据（即为新的字段提供相应合理的值），一个用于更新单个发票数据，另一个用于更新单个发票项。该解决方案需要修改所有的.go文件，其中 `jsn.go`、`xml.go` 和 `txt.go` 文件需要大改。总之，这些更改一起总共需要添加大约150行的代码。

[1].由于不存在一个幻数的全局库，我们无法确定任何特定的幻数之前是否已经被使用过。

[2].关于该格式更详细的描述请参考文档见 golang.org/pkg/encoding/gob/。Rob Pike也写了一篇有趣的关于gob格式的文章见blog.golang.org/2011/03/gobs-of-data.html。

[3].文件的权限通常以十进制数表示，以0开头。值为0666的文件是任何人都可以读写的——然而，掩码0022（一种常用的设置）可将该文件的权限设置为0644，因此将其设备成其创建者可读写而任何其他人只能读而不能写。

[4].如前所述，文件的权限标记通常写成八进制数字的形式，0666 对于文件来说是比较合理的。对于目录，0755 是比较合理的。

第9章 包

Go语言的标准库里包含大量的包，提供了大量、广泛而且富有创意的各种功能。另外，在Go Dashboard (godashboard.appspot.com/project) 上还有很多第三方的包可以使用。

除了这些，我们还能创建自己的包，安装到标准库里面，或者只是保留在我们自己的Go语言目录树里，也就是GOPATH路径。

在这一章我们将描述如何创建和导入一个自定义的包或者第三方的包，然后简略地了解下gc编译器的一些命令行参数，最后，我们来看一下Go语言的标准库，避免重复造轮子。

9.1 自定义包

到目前为止，我们见过的所有例子都是以一个包的形式存在的，也就是main包。在Go语言里，允许我们将同一个包的代码分隔成多个小块来单独保存，只需要将这些文件放在同一个目录即可。例如，第8章的invoicedata例子，虽然有6个独立的文件（invoicedata.go、gob.go、inv.go、jsn.go、txt.go和xml.go），但是每个文件的第一条语句都是包（main），表明它们都是同属于一个包的，也就是main包。

对于更大的应用程序，我们可能更喜欢将它的功能性分隔成逻辑的单元，分别在不同的包里实现。或者将一些应用程序通用的那一部分剥离出来。Go语言里并没有限制一个应用程序能导入多少个包或者

一个包能被多少个应用程序共享，但是将这些应用程序特定的包放在当前应用程序的子目录下和放在GOPATH源码目录下是不大一样的。我们所说的GOPATH源码目录是一个叫src的目录，每一个在GOPATH环境变量里的目录都应该包含这个目录，因为Go语言的工具链就是要求这样做的。我们的程序和包都应该在这个src目录下的子目录里。

我们也可以将我们自己的包安装到Go语言包目录树下，也就是GOROOT下，但是这样没有什么好处而且可能会不太方便，因为有些系统是通过包管理系统来安装Go语言的，有些是通过安装包，有些是手动编译的。

9.1.1 创建自定义的包

我们创建的自定义的包最好就放在GOPATH的src目录下（或者GOPATH src的某个子目录），如果这个包只属于某个应用程序，可以直接放在应用程序的子目录下，但如果我们希望这个包可以被其他的应用程序共享，那就应该放在GOPATH的src目录下，每个包单独放在一个目录里，如果两个不同的包放在同一个目录下，会出现名字冲突的编译错误。

作为惯例，包的源代码应放在一个同名的文件夹下面。同一个包可以有任意多个文件，文件的名字没有任何规定（但后续名必须是.go），在这本书里我们假设包名就是.go的文件名（如果一个包有多个.go文件，则其中会有一个.go文件的文件名和包名相同）。

第1章（1.5节）的stacker例子由一个主程序（在stacker.go文件里）和一个自定义的stack包（在文件stack.go里）组成，源码目录的层次结构如下：

aGoPath/src/stacker/stacker.go

aGoPath/src/stacker/stack/stack.go

`GOPATH`环境变量是由多个目录路径组成且路径之间以冒号（Windows上是分号）分隔开的字符串，这里的GOPATH路径集合中的其中一个路径。

我们在stacker目录里执行`go build`命令，就会得到一个stacker的可执行文件（在Windows系统上是`stacker.exe`）。但是，如果我们希望生成的可执行文件放到`GOPATH`的bin目录里，或者想将`stacker/stack`包共享给其他的应用程序使用，这就必须使用`go install`来完成。

当执行`go install`命令创建stacker程序时，会创建两个目录（如果不存在就会创建）：`aGoPath/bin`和，前者包含了stacker可执行文件，后者包含了stack包的静态库文件（至于linux_amd64等会根据不同的系统和硬件体系结构而变化，例如在32位的Windows系统上是windows_386）。

需要在stacker程序中使用stack包时，在程序源文件中使用导入语句`import "stacker/stack"`即可，也就是绝对路径（Unix风格）去除GOPATH下，都可以被别的程序或者包导入，`GOPATH`下的包没有共享和专用之分。

又比如第6章（6.5.3节）实现的有序映射是在omap包里，它被设计为可由多个程序使用。为了避免包名的冲突，我们在`GOPATH`（如果`GOPATH`有多个路径，任意一个路径都可以）路径下创建了一个具有唯一名字（这里用了域名）的目录，结构如下：

`aGoPath/src/qtrac.eu/omap/omap.go`

这样其他的程序，只要它们也在某个`GOPATH`目录下面，都可以通过使用`import "qtrac.eu/omap"`来导入这个包。如果我们还有其他的包需要共享，则将它们放到

当使用`go install`安装omap包的时候，它创建了

omap包的静态库文件，其中linux_amd64是根据不同的系统和硬件体系结构而变化的。

如果我们希望在一个包里创建新的包，例如，在my_package包下面创建两个新的包pkg1和pkg2，可以这么做：在aGoPath/src/my_package下创建两个子目录，例如aGoPath/src/my_package/pkg1和aGoPath/src/my_package/pkg2，对应的包文件是aGoPath/src/my_package/pkg1/pkg1.go和aGoPath/src/my_package/pkg2/pkg2.go。之后，假如想导入pkg2，使用import my_package/pkg2即可。Go语言标准库的源码树就是这样的结构。当然，my_package目录可以有它自己的包，如aGoPath/src/my_package/my_package.go文件。

Go语言中的包导入的搜索路径是首先到GOROOT（即\$GOROOT/pkg/\${GOOS}_\${GOARCH}，比如/opt/go/pkg/linux_amd64），然后是GOPATH环境变量下的目录。这就意味这可能会有名字冲突。最简单的方法就是确保GOPATH里包含的每个路径都是唯一的，例如之前我们以域名来作为omap的包的目录。

在Go程序里使用标准库里的包和使用GOPATH路径下的包是一样的，下面几个小节我们来讨论一些平台特定的代码。

9.1.1.1 平台特定的代码

在某些情况下，我们必须为不同的平台编写一些特定的代码。例如，在类Unix的系统上，通常shell都支持通配符（也叫globbing），所以在命令行输入*.txt，程序就能够从os.Args[1:]切片里读取到比如["README.txt", "INSTALL.txt"]这些值。但是在Windows平台上，程序只会接收到["*.txt"]，我们可以使用filepath.Glob()函数来实现通配符的功能，但是这只需要在Windows平台上使用。

那如何决定什么时候才需要使用filepath.Glob()函数呢，使用if runtime.GOOS == "windows" {...}即可，这也是本书中使用最广的方

法，例如cgrep1/cgrep1.go程序等等。另一种办法就是使用平台特定的代码来实现，例如，cgrep3程序有3个文件，cgrep.go、util_linux.go、util_windows.go，其中util_linux.go定义了这么一个函数：

```
func commandLineFiles(files []string) []string { return files }
```

很明显，这个函数并没有处理文件名通配，因为在类 Unix 系统上没必要这么做。而util_windows.go文件则定义了另一个同名的函数。

```
func commandLineFiles(files []string) []string {  
    args := make([]string, 0, len(files))  
    for _, name := range files {  
        if matches, err := filepath.Glob(name); err != nil {  
            args = append(args, name) // 无效模式  
        } else if matches != nil { // 至少有一个匹配  
            args = append(args, matches...)  
        }  
    }  
    return args  
}
```

当我们使用 go build 来创建 cgrep3 程序时，在 Linux 机器上 util_linux.go 文件会被编译而 util_windows.go 则被忽略，而在 Windows 平台恰好相反，这样就确保了只有一个commandLineFiles()函数被实际编译了。

在 Mac OS X 系统和FreeBSD 系统上，既不会编译 util_linux.go 也不会编译util_windows.go，所以go build会返回失败。但是我们可以创建一个软链接或者直接复制 util_linux.go 到 util_darwin.go 或者 util_freebsd.go，因为这两个平台的shell也是支持通配符的，这样就能正常构建Mac OS X和FreeBSD平台的程序了。

9.1.1.2 文档化相关的包

如果我们想共享一些包，为了方便其他的开发者使用，需要编写足够的文档才行。Go语言提供了非常方便的文档化工具 **godoc**，可以在命令行显示包的文档和函数，也可以作为一个Web服务器启动，如图9-1所示 [1]。godoc会自动搜索GOPATH路径下的所有包并将它显示出来，如果某些包不在GOPATH路径下，可以使用 **-path** 参数（除此之外还要有 **-http** 参数）来指定（我们在关于Go语言官方文档的部分讨论了godoc，参见1.1节）。

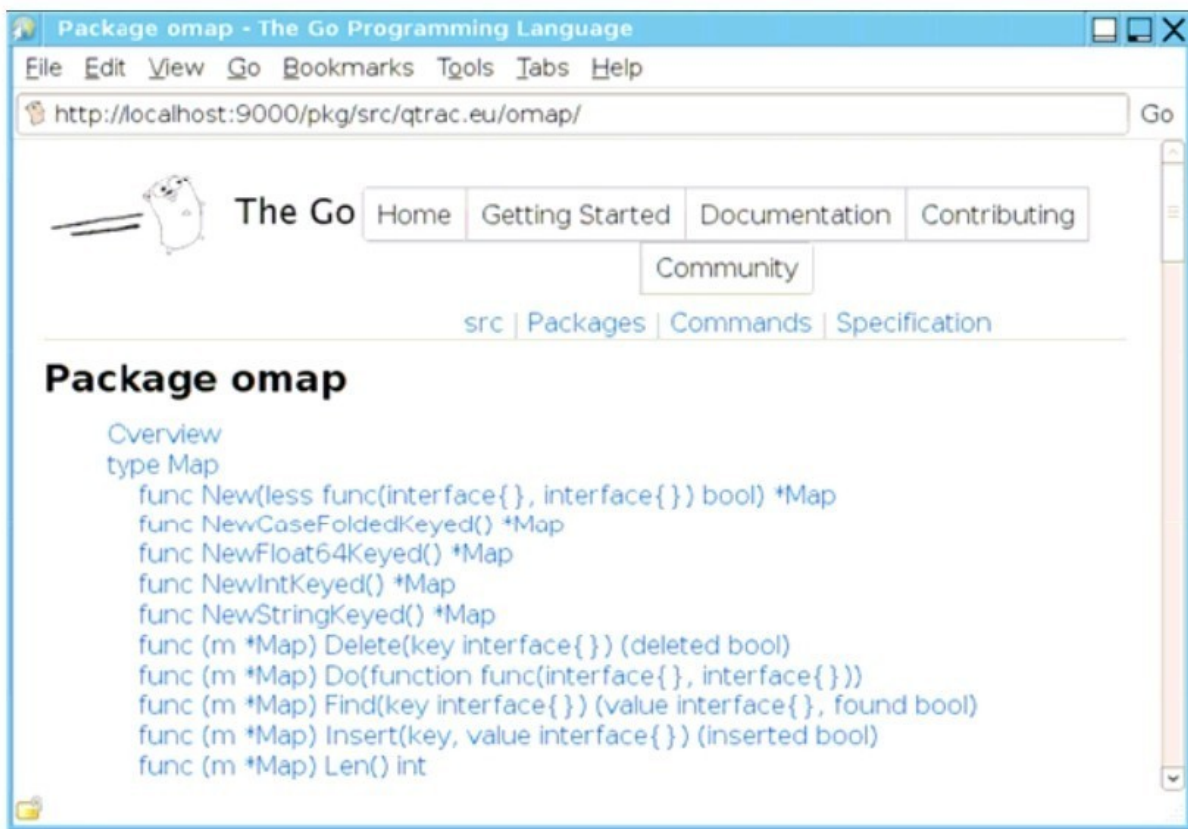


图9-1 omap包的文档

好的文档应该怎么写，这是一个一直争论不休的问题，因此在这一节我们只是纯粹地来了解一下Go语言的文档化机制。

在默认情况下，只有可导出的类型、类、常量和变量才会在godoc里出现，因此全部这些内容应该添加合适的文档。文档都是直接包含

在源文件里。这里以omap包为例（omap包我们之前已经在第6章讲过了）。

```
// Package omap implements an efficient key-ordered map.
//
// Keys and values may be of any type, but all keys must be comparable
// using the less than function that is passed in to the omap.New()
// function, or the less than function provided by the omap.New*()
// construction functions.
```

```
package omap
```

对于一个包来说，在包声明语句（package）之前的注释被视为是对包的说明，第一句是一行简短的描述，通常以句号结束，如果没有句号，则以换行符号结束。

```
// Map is a key-ordered map.
```

```
// The zero value is an invalid map! Use one of the construction
functions
```

```
// (e.g., New()), to create a map for a specific key type.
```

```
type Map struct {
```

可导出类型声明的文档必须紧接在该类型声明之前，而且必须总是描述该类型的零值是否有效。

```
// New returns an empty Map that uses the given less than function to
```

```
// compare keys. For example:
```

```
//     type Point { X, Y int }
```

```
//         pointMap := omap.New(func(a, b interface{}) bool {
```

```
//              $\alpha, \beta := a.(Point), b.(Point)$ 
```

```
//             if  $\alpha.X \neq \beta.X$  {
```

```
//                 return  $\alpha.X < \beta.X$ 
```

```
//             }
```



```
//          return  $\alpha.Y < \beta.Y$ 
//      })
```

```
func New(less func(interface{}, interface{}) bool) *Map {
```

函数或者方法的文档必须紧接在它们的第一行代码之前。上面这个例子是对 `omap` 包的 `New()` 构造函数的注释。

图9-2以Web的方式展示了一个函数的文档是什么样的，同时注释里缩进的文本会被视为代码显示在HTML页面上。但是在我写这本书的时候，`godoc`还不支持任何标记，例如**bold**、*italic*、[links](#)等。

```
// NewCaseFoldedKeyed returns an empty Map that accepts case-
insensitive
```

```
// string keys.
```

```
func NewCaseFoldedKeyed() *Map {
```

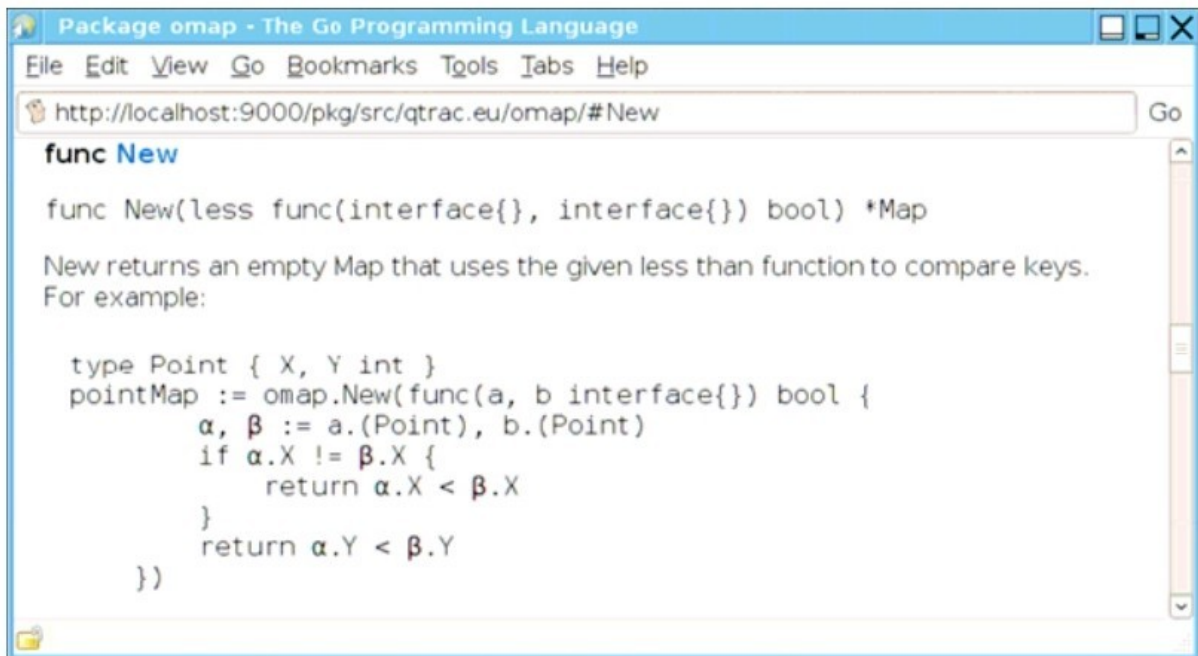


图9-2 `omap`包中`New()`函数的文档

上面这段文档是描述了一个出于便捷方面考虑而提供的辅助构造函数，基于一个预定义的比较函数。

```
// Insert inserts a new key-value into the Map and returns true; or
// replaces an existing key-value pair's value if the keys are equal and
// returns false. For example:
//      inserted := myMap.Insert(key, value).
func (m *Map) Insert(key, value interface{}) (inserted bool) {
```

这段是Insert()方法的文档。可以注意到Go语言里的文档描述通常是以函数或者方法的名字开头的，这是一个惯例，还有（这个不是惯例）就是在文档中引用函数或者方法时不使用圆括号。

9.1.1.3 包的单元测试和基准测试相关的包

Go语言标准库的testing包对单元测试提供了很好的支持。对一个包进行单元测试是一件很简单的事情，只需要在这个包的根目录下创建一个测试文件即可。单元测试的文件名格式为包名_test.go。例如，omap包的单元测试文件为omap_test.go。

在我们这本书里，单元测试文件都是放在一个独立的包（如omap_test）下面，然后导入需要被测试的包、testing包以及其他一些测试依赖的包。这实际上限制我们只能进行黑盒测试。但是也有些Go语言程序员会倾向于白盒测试。这很容易做到，只需将测试文件和包源代码放在一起即可，这种情况下我们不需要导入被测试的包，而且还可以测试那些非导出的数据类型，甚至为它们增加一些新的方法以方便测试。

单元测试文件比较特殊的一点是，它没有main()函数。取而代之的是，它有一些以Test开头的函数，并且必须只有一个*testing.T类型的参数，没有返回值。我们还可以增加任意其他的辅助函数，当然，这些函数不能以Test开头。

```
func TestStringKeyOMapInsertion(t *testing.T) {
    wordForWord := omap.NewCaseFoldedKeyed()
```

```

    for _, word := range []string{ " one " , " Two " , " THREE " , "
four " , " Five " } {
        wordForWord.Insert(word, word)
    }
    var words []string
    wordForWord.Do(func(_, value interface{}) {
        words = append(words, value.(string))
    })
    actual, expected := strings.Join(words, " "), "
FivefouroneTHREETwo "
    if actual != expected {
        t.Errorf( " %q != %q " , actual, expected)
    }
}

```

这是 `omap_test.go` 文件里的一个单元测试。首先创建一个空的 `omap.Map`，然后插入一些字符串类型的键和值（注意键名是区分大小写的）。然后使用 `Map.Do()` 方法遍历，将得到的每个值追加到一个字符串切片里去。最后，将这个字符串切片组合成一个字符串，检查结果是否是我们所期望的。如果结果不对，则调用 `testing.T.Errorf()` 方法报告详细的失败的原因。如果错误或者失败方法没有被调用，我们就可以假定测试已经通过了。

测试通过的结果类似如下。

```

$ go test
ok      qtrac.eu/omap
PASS

```

如果测试的时候使用 `-test.v` 选项，则会输出更详细的信息。

```

$ go test -test.v

```

```
ok          qtrac.eu/omap
=== RUN TestStringKeyOMapInsertion-4
--- PASS: TestStringKeyOMapInsertion-4 (0.00 seconds)
=== RUN TestIntKeyOMapFind-4
--- PASS: TestIntKeyOMapFind-4 (0.00 seconds)
=== RUN TestIntKeyOMapDelete-4
--- PASS: TestIntKeyOMapDelete-4 (0.00 seconds)
=== RUN TestPassing-4
--- PASS: TestPassing-4 (0.00 seconds)
PASS
```

如果测试不通过，会得到如下信息（这里我们人为地修改了常量的字符串值，强制它失败），如果指定了`-test.v`选项，得到的信息可能更多。

```
$ go test
FAIL          qtrac.eu/omap
--- FAIL: TestStringKeyOMapInsertion-4 (0.01 seconds)
        omap_test.go:35:  "  FivefouroneTHREETwo  "  !=  "
        FivefouroneTHREEToo "
FAIL
```

另外，这个例子里用到了`Errorf()`方法，`testing`包的`*testing.T`还有很多其他的方法可以使用，如`testing.T.Fail()`、`testing.T.Fatal()`等。利用这些方法我们可以实现测试的调试级别。

此外，`testing`包还支持基准测试，和其他的测试函数一样，基准测试也是放在`package_test.go`文件里的，唯一不同的就是基准测试的函数名必须以`Benchmark`开头，并且必须有一个`*testing.B`类型的参数，没有返回值。

```
func BenchmarkOMapFindSuccess(b *testing.B) {
```

```

b.StopTimer() // Don't time creation and population
intMap := omap.NewIntKeyed()
for i := 0; i < 1e6; i++ {
    intMap.Insert(i, i)
}
b.StartTimer() // Time the Find() method succeeding
for i := 0; i < b.N; i++ {
    intMap.Find(i % 1e6)
}
}

```

函数一开始是就执行 `b.StopTimer()` 来停止计时器，因为我们不希望将创建和生成 `omap.Map` 的时间也计算在内。我们创建一个空的 `omap.Map`，然后插入一百万条记录。

默认情况下 `go test` 不会执行基准测试，所以如果我们需要基准测试的时候必须指定 `-test.bench` 选项，还需要有一个正则表达式字符串，来匹配我们需要执行的基准测试函数名。例如 `*` 表示所有的基准测试函数都会被执行（只有一个.也行）。

```

$ go test -test.bench=.
PASS          qtrac.eu/omap
PASS
BenchmarkOMapFindSuccess-4    1000000    1380ns/op
BenchmarkOMapFindFailure-4   1000000    1350ns/op

```

从这个结果我们可以看出，两个函数都遍历了一百万次，还给出了每次操作的时间消耗。至于遍历多少次是由 `go test` 来决定的，也就是 `b.N`，不过我们可以使用 `-test.benchtime` 选项来指定我们希望每个基准测试的执行时间为多少秒。

本书中还有其他的一些例子，也是使用`package_test.go`作为测试文件的。

9.1.2 导入包

Go语言允许我们对导入的包使用别名来标识。这个特性是非常方便和有用的，例如，可以在不同的实现之间进行自由的切换。举个例子，假如我们实现了`bio`包的两个版本`bio_v1`和`bio_v2`，现在在某个程序里使用了`import bio "bio_v1"`，如果需要切换到另一个版本的实现，只需要将`bio_v1`改成`bio_v2`即可，即`import bio "bio_v2"`，但是需要注意的是，`bio_v1`和`bio_v2`的API必须是相同的，或者`bio_v2`是`bio_v1`的超集，这样其余所有的代码都不需要做任何改动。另外，最好就不要对官方标准库的包使用别名，因为这样可能会导致一些混淆或激怒后来的维护者。

我们之前在第5章提到过（见5.6.2节），当导入一个包时，它所有的`init()`函数就会被执行。有些时候我们并非真的需要使用这些包，仅仅是希望它的`init()`函数被执行而已。

举个例子，如果我们需要处理图像，通常会导入 Go 标准库支持的所有相关的包，但是并不会用到这些包的任何函数。下面就是`imtag1.go`程序的导入语句部分。

```
import (  
    "fmt"  
    "image"  
    "os"  
    "path/filepath"  
    "runtime"  
    _ "image/gif"
```

```
    _ "image/jpeg "  
    _ "image/png "  
)
```

这里导入了image/gif、image/jpeg和image/png包，纯粹是为了让它们的init()函数被执行（这些init()函数注册了各自的图像格式），所有这些包都以下划线作为别名，所以Go语言不会发出导入了某个包但是没有使用的警告。

9.2 第三方包

Go语言的工具链几乎贯穿全书了，我们使用它来创建程序和包，如omap包等。除此之外，我们还可以用来下载、编译和安装第三方的包。当然，前提必须是我们的计算机能够连接网络。godashboard.appspot.com/project上面维护了一系列第三方的包。（另外一种方法就是通过下载源码，通常是通过版本控制系统来下载，然后本地编译。）

需要安装Go Dashboard的包的话，首先点击它的链接到包的主页，然后找到有go get命令的地方，通常那就是介绍如何下载和安装包的了。

举个例子，我们点击Go Dashboard页面上的freetype-go.googlecode.com/hg/ freetype 链接，然后它会将我们带到code.google.com/p/freetype-go/主页，这个页面上有如何安装的相关介绍，在我写这本书的时候，这个命令是go get freetype-go.googlecode.com/hg/freetype。

毕竟这个包是来自于第三方的，go get 还必须将它安装到我们计算机上的某个地方。默认情况下会安装到GOPATH环境变量的第一个路

径，如果没法将这个包保存到那里，就自动安装到GOROOT目录。如果我们想强制go get默认使用GOROOT目录，可以在go get运行之前清空GOPATH环境变量中的路径集合。

执行go get 之后，就自动开始下载、创建和安装包了。如果想了解最新安装的包的文档，可以以Web服务的方式来运行godoc，例如godoc -http=:8000，这样就可以查看这个包的文档了。

为了避免名字上的冲突，第三方的包通常使用域名来确保唯一性。举个例子，假如我们想使用FreeType这个包的话，可以这样导入：

```
import "freetype-go.googlecode.com/hg/freetype"
```

当然，使用这个包的函数我们只需要最后一部分即可，也就是freetype，比如font, err :=freetype.ParseFont(fontdata)。如果很不幸地连最后一部分也产生名字冲突了，我们还可以使用别名，例如，import ftype "freetype-go.googlecode.com/hg/freetype"，然后在我们的代码里这样写font, err := ftype.ParseFont(fontdata)。

第三方的包通常都可以在Go 1上使用，但是有些需要更新的Go版本，或者提供多个版本的下载。比如，有一些需要在最新的开发版上才能使用。通常情况下，最好只使用稳定版的Go（目前是Go 1）和与该版本兼容的第三方包。

9.3 Go命令行工具简介

安装Go的gc编译器自然也就包括了编译器和连接器（6g、6l等），还有其他的一些工具。最常用的就是go，既可以用来创建我们自己的程序和包，又可以下载和安装第三方的程序和包，还可以用来执行单元测试和基准测试，如我们之前在9.1.1.3节中见到的一样。如果

需要更多的帮助可以使用 `go help` 命令，`go help` 会显示一个命令列表，当然文档化的 `godoc` 工具也在其中。

除了我们这本书所用到的工具，还有其他的一些工具和 `go tool` 命令，这里我们会介绍一些。其中一个就是 `go vet` 命令，它可以检查Go程序的一些简单错误，特别是 `fmt` 包的打印函数。

另一个命令就是 `go fix`，有时候 Go 语言的新发行版会包含一些语言上的变更，或者更多的是标准库API的修改，这样会导致我们写好的代码编译不过。这种情况下可以在我们代码的根目录下执行 `go fix` 命令来进行自动的升级。我们强烈推荐你使用版本控制系统来管理你的 `.go` 文件，这样所有的修改都会记录下来，或者在运行 `go fix` 之前至少也做一个备份。这样做的原因是 `go fix` 有可能会破坏我们现有的代码，如果真的发生了，至少我们还可以恢复它。我们还可以使用带有 `-diff` 选项的 `go fix` 命令，这样可以看到 `go fix` 将要修改哪些地方，但并不会真地修改它们。

最后一个要介绍的命令就是 `gofmt`。它能以一种标准化的方式来格式化Go代码，这也是Go的开发者强烈推荐使用的。使用 `gofmt` 的最大好处就是，你不需要考虑哪种编排方式最好，`gofmt` 可以让你所有的代码看起来都是同一种风格。我们这本书所有的代码都是经过 `gofmt` 格式化的，不过超过75个字符的代码行会被自动折行以适合本书的页宽。

[9.4 Go标准库简介](#)

Go标准库里包含大量的包，功能非常丰富。我们这里只是做一个很简单的介绍，因为标准库的内容是随时都有可能被改动的，最好的方式就是浏览官方在线版的标准库（golang.org/pkg/），或者使用本地

的godoc，这两种方式都能看到最新的信息并且可以更好地理解每个包提供了什么样的功能。

其中 `exp` (`experimental`) 包包含一些将来有可能（也可能不）被增加到标准库里面去的实验性包，所以除非我们想参与标准库的开发，否则就不要使用这个包。在以编译源代码方式安装Go的时候通常会带有这个`exp`包，但通常不会被包含在Go语言安装包中。这里介绍的所有包都可以使用，尽管在我写这本书的时候有些包还不是很完整。

9.4.1 归档和压缩包

Go 语言提供了用于读写tar包文件和.zip文件的包 `archive/tar` 和 `archive/zip`，如果需要压缩tar包，还可以使用 `compress/gzip` 和 `compress/bzip2`，这本书第8章的`pack`和`unpack`例子就涵盖了这些功能的用法（参见8.2节）。

其他的压缩格式也支持，例如LZW格式。`compress/lzw`包主要是用来处理.tiff图像和.pdf文件。

9.4.2 字节流和字符串相关的包

`bytes`和`strings`这两个包有很多函数是一样的，只不过前者是处理`[]byte`类型的值，而后者是处理`string`类型的值。对字符串来说，`strings`包提供了大部分常用的功能，例如查找子串、替换子串、切割字符串、过滤字符串、改变大小写（参见3.6.1节），等等。还有，利用`strconv`包可以很方便地将数值和布尔型类型的值转换成字符串，反过来也可以（参见3.6.2节）。

`fmt`包提供了很多非常有用的打印函数和扫描函数。打印函数在第3章已经讲过，扫描函数在表8-2也列出来了，后面还紧接着有一些用例。

`unicode` 包可以用来判断字符的属性，比如一个字符是否是可打印的，或者是否是一个数字（参见3.6.4节）。`unicode/utf8`和`unicode/utf16`这两个包主要用来编码和解码 `rune`（也就是Unicode码点），其中`unicode/utf8`参见3.6.3节，部分还在第8章的`utf16-to-utf8`练习中出现过。

`text/template`和`html/template`包可以用来创建模板，借助模板可以很容易地输出比如HTML等这些文档，只要将一些数据填充进去即可。下面是一个非常简单的`text/template`包的例子。

```
type GiniIndex struct {
    Country string
    Index float64
}

gini := []GiniIndex{{ " Japan " , 54.7}, { " China " , 55.0}, { " U.S.A.
" , 80.1}}

giniTable := template.New( " giniTable " )
giniTable.Parse(
    '<TABLE>' +
        '{{range.}}' +
        '{{printf    "    <TR><TD>%s</TD><TD>%.1f%%</TD></TR>
" .Country.Index}}'+
        '{{end}}' +
        '</TABLE>')

err := giniTable.Execute(os.Stdout, gini)
```

```
<TABLE>
<TR><TD>Japan</TD><TD>54.7%</TD></TR>
<TR><TD>China</TD><TD>55.0%</TD></TR>
<TR><TD>U.S.A.</TD><TD>80.1%</TD></TR>
</TABLE>
```

`template.New()` 函数根据给定的模板名创建了一个新的 `*template.Template` 类型的值。模板名用于在模板嵌套的时候标识特定模板。`template.Template.Parse()` 函数解析一个模板（通常是一个 `.html` 文件）以备使用。`template.Template.Execute()` 函数执行一个模板，并从它的第二个参数读取数据，最后将结果发送到给定的 `io.Writer` 里去。在这个例子中，从 `gini` 切片读取数据，`gini` 是一个 `GiniIndex` 结构体，然后将结果输出到 `os.Stdout`。（为了清晰起见，我们将输出结果分成一行一行地显示。）

模板里的所有动作都是在一个双大括号 `{{...}}` 里的，还可以使用 `{{range}}...{{end}}` 来遍历一个切片里的所有项，这里我们使用点号 `(.)` 来表示 `GiniIndex` 切片里的每一项，也就是，这个点号可以理解为当前的项。我们可以使用字段名来访问一个结构体的可导出的字段，当然，点号表示当前的项。`{{printf}}` 动作和 `fmt.Printf()` 函数是一样的，只是使用空格符号来表示圆括号和参数分隔符。

`text/template` 和 `html/template` 包支持原始的模板语言和很多动作，包括遍历和条件分支，支持变量和方法调用，还有许多。此外，`html/template` 包还可以安全地防止代码注入。

9.4.3 容器包

Go语言里的切片是一种非常高效的集合类型，但有些时候自定义一些特别的集合类型是有用的，或者是必需的。大部分情况下用内置的 `map` 就能解决很多问题，不过Go语言还是提供了 `container` 包来支持更多的容器类型。

我们可以使用 `container/heap` 包提供的函数来操作一个堆，前提是这个堆上的元素的类型必须满足 `heap` 包中 `heap.Interface` 接口的定义。堆（严格来说是最小堆）维护了一个有序数组，保证堆上的第一个元素

肯定是最小的（如果是最大堆，则第一个元素是最大的），这是大家熟知的堆的特性。heap.Interface 接口嵌入了 sort.Interface接口，并增加了 Push()和Pop()方法（其中 sort.Interface 我们在 4.2.4 节和5.7节讲解过）。

要创建一个满足heap.Interface接口定义的堆还蛮简单的，这是一个例子。

```
ints := &IntHeap{5, 1, 6, 7, 9, 8, 2, 4}
heap.Init(ints) // 将其转换成堆
ints.Push(9)    // IntHeap.Push()并未保持堆的属性
ints.Push(7)
ints.Push(3)
heap.Init(ints) // 堆被打破后必须重新将其转换成堆
for ints.Len() > 0 {
    fmt.Printf( " %v ", heap.Pop(ints))
}
fmt.Println() // 打印1 2 3 4 5 6 7 7 8 9 9
```

下面是一个完整的自定义堆实现。

```
type IntHeap []int
func (ints *IntHeap) Less(i, j int) bool {
    return (*ints)[i] < (*ints)[j]
}
func (ints *IntHeap) Swap(i, j int) {
    (*ints)[i], (*ints)[j] = (*ints)[j], (*ints)[i]
}
func (ints *IntHeap) Len() int {
    return len(*ints)
}
```

```

func (ints *IntHeap) Pop() interface{} {
    x := (*ints)[ints.Len()-1]
    *ints = (*ints)[:ints.Len()-1]
    return x
}
func (ints *IntHeap) Push(x interface{}) {
    *ints = append(*ints, x.(int))
}

```

大多时候实现一个这样的堆能够解决很多问题了。为了让代码的可读性更高一些，我们将这个堆定义为`IntHeap struct { ints []int }`，这样我们就可以在方法里引用`ints.ints`而不是`*ints`。

`container/list` 包提供了双向链表的支持，可以将一个值以`interface{}`的类型添加到链表里去。从`list`里得到的项的类型是`list.Element`，可以使用`list.Element.Value`来访问我们添加进去的值。

```

items := list.New()
for _, x := range strings.Split( " ABCDEFGH " , " " ) {
    items.PushFront(x)
}
items.PushBack(9)
for element := items.Front(); element != nil; element = element.Next() {
    switch value := element.Value.(type) {
    case string:
        fmt.Printf( " %s " , value)
    case int:
        fmt.Printf( " %d " , value)
    }
}

```

```
fmt.Println() // 打印H G F E D B A 9
```

在这个例子里我们将8个字母依次添加到链表里的最前端，并同时添加一个int型值在最后端。然后我们遍历列表里的元素将每个元素的值打印出来，这里我们不需要使用类型开关，因为我们可以使用 `fmt.Printf(" %v " , element.Value)`。但如果我们不仅仅是只打印出它的值，还有其他的用途，这时候就得做类型开关。当然，如果所有的类型都是一样的话，我们可以使用一个类型断言，例如 `element.Value.(string)`可以用来判断字符串。（关于类型开关我们在5.2.2.2节讲过，类型断言则在5.1.2节。）

除了上面我们刚介绍过的，`list.List`类型还提供了很多方法，包括 `Back()`、`Init()`（用来清空一个列表）`InsertAfter()`、`InsertBefore()`、`Len()`、`MoveToBack()`、`PushBackList()`（将一个列表添加到另一个列表的末尾）。

标准库还提供了`container/ring`包，实现了一个环形的单向列表。

[2]

因为这些容器类型的所有数据都是保存在内存的，如果数据量很大，可以考虑使用标准库里提供的`database/sql`包来将数据存储到数据库里，`database/sql`实现了一个SQL数据库的通用接口。实际使用的时候还必须安装数据库的相关驱动才行，这些和很多其他的容器包一样，可以从Go Dashboard（godashboard.appspot.com/project）里下载。还有就是之前我们看到的，本书包含了一个有序映射`omap.Map`类型，它基于左倾红黑树（left-leaning red-black Tree，参见6.5.3节）。

9.4.4 文件和操作系统相关的包

标准库里提供了很多包来支持文件和目录相关的处理，以及一些和操作系统交互的系统调用。大多数情况下这些包都提供了一个平台

无关的抽象层，方便我们写出跨平台的代码。

`os` (`operating system`) 包提供了很多操作系统相关的函数，例如更改当前工作目录，修改文件权限和所有者，读取和设置环境变量，还有创建、删除文件和目录。另外，`os`包还提供了创建和打开文件（`os.Create`和`os.Open`）、获取文件属性（例如通过`os.FileInfo`类型）相关的函数，这些在我们之前的章节里都全部用过了（参见7.2.5、第8章）。

一旦文件被打开了，特别是文本文件，经常需要用到一个缓冲区来访问它的内容（例如读取一行字符串而不是一个字节切片）。我们可以通过使用`bufio`包来实现这个功能，之前就有好些例子是这么用的了。除了使用`bufio.Reader`和`bufio.Writer`来读写字符串，我们还可以读取（或者倒退）`rune`、单个字节、多个字节，还可以写一个`rune`、一个或者多个字节。

`io`包提供了大量的与输入输出相关的函数，用来处理`io.Reader`和`io.Writer`。（`*os.File`类型的值能同时满足这两个接口的定义。）比如我们可以使用`io.Copy()`函数来将数据从一个`reader`复制到一个`writer`里去（参见8.2.1节）。另外，这个包还能用来创建内存中的同步管道。

`io/ioutil`包提供了一些高级的辅助函数，例如，可以使用`ioutil.ReadAll()`函数来将一个`io.Reader`的所有数据读取到一个`[]byte`中。`ioutil.ReadFile()`函数也是同样的功能，只是参数必须是字符串，也就是文件名，而不是一个`io.Reader`。`ioutil.TempFile()`函数用来创建一个临时文件，返回`*os.File`类型的值，还有`ioutil.WriteFile()`函数可以将一个`[]byte`写到指定的文件里去。

`path`包用来操作Unix风格的路径，例如Linux和Mac OS X路径、URL路径、`git`引用、FTP文件，等等。还有`path/filepath`包提供了和`path`相同的函数（当然还有其他的），其目的是提供平台无关的路径处

理。这个包还提供了 `filepath.Walk()` 函数用来遍历读取一个给定路径下的所有文件和目录信息，如我们之前在7.2.5节看过的。

`runtime` 包含一些函数和类型，可以用来访问Go语言的运行时系统。大部分都是一些高级功能，很多时候我们用不到。不过有两个常量还是经常有用的，就是 `runtime.GOOS` 和 `runtime.GOARCH`，两个都是字符串，前者的值可能是“Darwin”、“freebsd”、“linux”或者“windows”，后者可以是“386”、“amd64”、“arm”等。`runtime.GOROOT()` 函数返回GOROOT环境变量的值（如果为空则返回Go安装环境的根目录），还有 `runtime.Version()` 函数返回当前Go语言的版本（字符串），之前我们在第7章还见过 `runtime.GOMAXPROCS()` 和 `runtime.NumCPU()` 函数来让Go语言使用机器上所有的处理器。

文件格式相关的包

Go语言标准库对文本文件（7位的ASCII编码、UTF-8或者UTF-16）的支持是非常优秀的，对二进制文件的支持也一样。Go语言提供了一些单独的包来处理JSON和XML文件，还包括它自己的高效简便的Go语言二进制格式（`gob`）。（这些格式，包括自定义的二进制格式，我们在第8章都讲过了。）

另外，Go语言还提供了 `csv` 包用来读取.csv文件（`csv` 是“comma-separated values”的缩写，即用逗号分隔的值）。这个包将文件当成是数据记录处理，也就是每一行被认为是一条记录，包含多个逗号分隔的字段值。这个包具有很大的通用性，例如，我们可以改变它的分隔符（用缩进或者其他的字符来取代逗号），还可以修改它对记录和字段的读写方式。

`encoding` 包里有好几个子包。其中一个就是 `encoding/binary`，我们用来读写二进制数据（参见 8.1.5 节）。还有其他的一些子包用来编码和解码一些比较常见的格式，例如，`encoding/base64` 包用来编码和解码URL，因为它经常会使用这种编码。

9.4.5 图像处理相关的包

Go语言的image包提供了一些高级的函数和数据类型，用来创建和保存图像数据。还包括一些用来对常见图像格式进行编解码的包，例如image/jpeg和image/png。其中一些我们在之前的章节已经讨论过了，比如9.1.2节和第7章的一个练习。

image/draw 包提供了一些基本的画图功能，例如我们之前在第6章见过的。第三方的freetype包为画图增加了一些功能，它可以使用特定的TrueType字体来画文本，还有freetype/raster包可以画行，画立方体，甚至是二次曲线。（我们在9.2节已经讲过如何获取和安装freetype包。）

9.4.6 数学处理包

math/big包可以创建没有大小限制（仅受内存大小限制）的整数（bit.Int）和有理数（big.Rat）。这些已经在之前的2.3.1.1节已经介绍过。math/big还提供了big.ProbablyPrime()函数。

math包提供了所有标准的数学处理函数，这些函数都是基于float64类型的，还有一些标准的常量，可以参见表2-8、表2-9和表2-10。

math/cmplx包提供了常见的复数相关函数（基于complex128类型），参见2.11节。

9.4.7 其他一些包

除了以上这些大致归类在一起的包以外，标准库也包含了一些相对有点独立的包。

crypto包提供了MD5、SHA-1、SHA-224、SHA-256、SHA-384、SHA-512等哈希算法。每一个哈希算法都以一个独立的包存在，例如

`crypto/sha512`。此外，我们还可以使用`crypto`包来进行加密和解密，例如使用AES算法、DES算法等，分别对应`crypto/aes`和`crypto/des`包。

`exec`包可用来运行外部的程序，当然这也可以使用`os.StartProcess()`函数来完成，但是`exec.Cmd`类型相对来说更加易用一些。

`flag`包是一个命令行解析器，可以接收X11风格的选项（例如，`-width`，而不是GNU风格，比如`-w`和`--width`）。这个包只能打印一条非常基本的用法帮助信息，并且没有提供任何输入值的合法检查相关的能力。（所以我们可以指定一个 `int` 选项，但无法指定什么范围的值是可接受的。）一些替代性的包在 Go Dashboard (godashboard.appspot.com/project) 上可以找到。

`log`包可以用来做日志记录（默认输出到 `os.Stdout`），可在程序退出或抛出异常的同时产生一条日志。可以使用`log.SetOutput()`函数将`log`包的输出目标更改成任意的`io.Writer`。日志的输出格式为时间戳和消息体，不想显示时间戳的话可以在第一条 `log`输出前调用`log.SetFlags(0)`。我们还可以使用`log.New()`来创建自定义的`logger`实例。

`math/rand`包可以生成伪随机数。`rand.Int()`返回随机的`int`型值，`rand.Intn(n)`返回一个在区间`[0,n)`范围内的`int`型值。`crypto/rand`包还提供了函数用来产生加解密用途的更高强度的伪随机数。

`regexp`包实现了一个非常快而强大的正则表达式引擎，支持RE2引擎语法。我们这本书就有好几个地方是用到这个包的，虽然为了不跑题我们只是简单地用了一下正则的功能，并没用到这个包全部的特性。这个包之前也介绍过了（参见3.6.5节）。

`sort`包可以很方便地对切片进行排序，包括`int`型的、`float64`型的和`string`类型的，并且提供了基于已排序的切片上的快速查找功能（基于二分查找）。还提供了通用的`sort.Sort()`函数和`sort.Search()`函数用来处理自定义的数据类型（参见4.2.4节的例子和表4-2以及5.6.7节）。

`time` 包主要包括计时和日期时间解析及格式化相关的函数。`time.After()`函数可以在指定时间间隔（就是我们传入的纳秒值参数）之后往该函数返回的通道里发送一个当时的时间值，我们在之前的例子里有介绍过它（参见7.2.2节）。`time.Tick()`和`time.NewTicker()`函数同样返回一个信道，不同的是我们可以定期地从这个信道里得到一个“嘀嗒”。`time.Time`结构实现了一些方法，例如，可以提供当前时间，格式化日期/时间为一个字符串，解析日期/时间。（我们在第8章看过`time.Time`的用法。）

9.4.8 网络包

Go标准库里还有很多包支持网络相关的编程。比如`net`包提供了一些通信相关的函数和数据类型，包括Unix域、网络套接字，以及TCP/IP和UDP通信等。还有一些函数用来进行域名解析。

`net/http`包使用了`net`包，提供了解析HTTP请求和响应的功能，并提供了一个基础的HTTP客户端。除此之外，还包含了一个易于扩展的HTTP服务器，就像我们在第2章和第3章的练习见到的那样。`net/url`包提供了URL解析和查询字符串的转义。

标准库里还有其他一些高级的网络包，其中一个就是`net/rpc`（远程过程调用），可以让客户端远程调用服务器上某些对象的导出方法。另一个就是`net/smtp`包（简单邮件传输协议），用来发送邮件。

9.4.9 反射包

反射包可以提供运行时的反射（`reflection`）功能（也叫类型检视，`introspection`），我们可以在运行时访问和操作任意类型的值。

这个包也提供了一些非常有用的功能。例如 `reflect.DeepEqual()`函数可以用来比较两个值，例如不能直接使用`==`或者`!=`操作符进行比较

的两个切片。

Go 语言里每一个值都有两个属性：实际的值和它的类型。
`reflect.TypeOf()`函数能告诉我们任意值的类型。

```
x := 8.6
y := float32(2.5)
fmt.Printf( " var x %v = %v\n " , reflect.TypeOf(x), x)
fmt.Printf( " var y %v = %v\n " , reflect.TypeOf(y), y)
var x float64 = 8.6
var y float32 = 2.5
```

这里我们使用反射功能输出两个`var`声明的浮点变量和它们的类型。

调用`reflect.ValueOf()`函数可以得到一个`reflect.Value`结构，这个结构保存了传入的值，但并不是传入的那个值本身。如果我们需要访问那个传入的值，必须使用 `reflect.Value`的方法。

```
word := " Chameleon "
value := reflect.ValueOf(word)
text := value.String()
fmt.Println(text)
```

```
Chameleon
```

`reflect.Value`类型实现了很多方法可以用来提取底层类型的实际值，包括 `reflect.Value.Bool()`、`reflect.Value.Complex()`、`reflect.Value.Float()`、`reflect.Value.Int()`和`reflect.Value.String()`。

同样，`reflect`包还可以用于集合类型，例如切片、映射以及结构体。它甚至能访问结构体的标签文本（`tag text`）。（如我们在第8章见到的，`json`和`xml`的编码器和解码器使用了这个功能。）

```
type Contact struct {
```

```

    Name string " check:len(3,40) "
    Id int " check:range(1,999999) "
}
person := Contact{ " Bjork " , 0xDEEDED}
personType := reflect.TypeOf(person)
if nameField, ok := personType.FieldByName( " Name " ); ok {
    fmt.Printf( " %q %q %q\n " , nameField.Type, nameField.Name,
nameField.Tag)
}

```

```

    " string " " Name " " check:len(3,40) "

```

如果一个被`reflect.Value`保存的底层类型的值是可设置的，那么我们可以改变它。这种可设置的能力可以使用`reflect.Value.CanSet()`来检查，它返回一个`bool`类型的值。

```

presidents := []string{ " Obama " , " Bushy " , " Clinton " }
sliceValue := reflect.ValueOf(presidents)
value = sliceValue.Index(1)
value.SetString( " Bush " )
fmt.Println(presidents)

```

```

[Obama Bush Clinton]

```

尽管在Go语言里字符串是不可修改的，但在一个`[]string`里任意一个项都可以被其他的字符串替换，这也是我们这里所做的。（自然地，在这个例子里我们可以更直接地使用`presidents[1] = " Bush "`来修改它，没必要使用反射。）

因为无法修改不可修改的值的內容，我們可以在得到原始值的地址后直接用另一個值來替換它。

```
count := 1
if value = reflect.ValueOf(count); value.CanSet() {
    value.SetInt(2) // 會拋出異常，不能設置一個int值
}
fmt.Print(count, " ")
value = reflect.ValueOf(&count)
// 不能調用SetInt，因為值是一個*int而不是int
pointee := value.Elem()
pointee.SetInt(3) // 成功。可以替換一個指針指向的值
fmt.Println(count)
```

13

從這段代碼可以看出，如果條件判斷失敗，則條件語句的主體部分不會被執行。儘管我們不能設置不可修改的值，如int、float64和string，但我們可以使用reflect.Value.Elem()方法來獲得reflect.Value的值，這樣實質上就允許我們修改一個指針指向的值了，也就是我們這塊代碼所做的。

同樣我們可以用反射來調用任意的函數和方法。下面就是一個例子，它調用了兩次一個自定義的TitleCase函數（代碼沒有展示出來），一次是直接調用，一次使用了反射。

```
caption := " greg egan's dark integers "
title := TitleCase(caption)
fmt.Println(title)
titleFuncValue := reflect.ValueOf(TitleCase)
values := titleFuncValue.Call([]reflect.Value{reflect.ValueOf(caption)})
```

```
title = values[0].String()
```

```
fmt.Println(title)
```

```
Greg Egan's Dark Integers
```

```
Greg Egan's Dark Integers
```

`reflect.Value.Call()` 方法传入和返回参数都是一个 `[]reflect.Value` 切片。在这个例子中我们传入了单个值（也就是一个长度为1的切片），并且得到一个结果值。

类似地，我们还能调用方法。实际上，我们甚至可以查询某个方法是否存在，进而再决定是否调用它。

```
a := list.New()                                // a.Len() == 0
b := list.New()
b.PushFront(1)                                // b.Len() == 1
c := stack.Stack{}
c.Push(0.5)
c.Push(1.5)                                    // c.Len() == 2
d := map[string]int{ " A " : 1, " B " : 2, " C " : 3} // len(d) == 3
e := " Four "                                  // len(e) == 4
f := []int{5, 0, 4, 1, 3}                      // len(f) == 5
fmt.Println(Len(a), Len(b), Len(c), Len(d), Len(e), Len(f))
```

```
0 1 2 3 4 5
```

这里创建了两个列表（使用 `container/list` 包），其中一个列表我们添加了一个项进去。我们也创建一个栈（使用在1.5节创建的自定义的 `stacker/stack` 包），然后往里面增加两个项。接着我们还创建了一个映射、一个字符串和一个 `int` 切片。所有这些值的长度都不同。

```
func Len(x interface{}) int {
    value := reflect.ValueOf(x)
```



```

switch reflect.TypeOf(x).Kind() {
case reflect.Array, reflect.Chan, reflect.Map, reflect.Slice,
reflect.String:
    return value.Len()
default:
    if method := value.MethodByName( " Len " ); method.IsValid() {
        values := method.Call(nil)
        return int(values[0].Int())
    }
}
panic(fmt.Sprintf( " '%v' does not have a length " , x))
}

```

这个函数返回传入值的长度，如果传入的这个值的类型不支持获得它的长度，就抛出一个异常。

首先我们获得一个`reflect.Value`值(后面会用到)。然后我们使用`switch`语句，根据这个值的`reflect.Kind`来进行条件处理。如果这个值的类型是支持内置 `len()` 函数的Go语言内置类型，我们直接调用`reflect.Value.Len()`函数。否则，这个类型要么不支持获得长度，要么没有实现`Len()`方法。我们使用`reflect.Value.MethodByName()`方法来获得某个指定的方法，可能得到一个不合法的`reflect.Value`值，如果这个方法是合法的，我们就调用它。这里我们不需要传入任何参数，因为`Len()`方法本身是不带参数的。

我们调用`reflect.Value.MethodByName()`方法得到的`reflect.Value`值同时保存了方法和值，所以当我们调用 `reflect.Value.Call()`时，这个值可以直接作为接收者（receiver）。

`reflect.Value.Int()`方法返回一个`int64`类型的值，我们还必须将它转换成`int`型以匹配`Len()`函数的返回值类型。

如果我们传入一个不支持内置 `len()` 函数的值，也没有 `Len()` 方法，那么就会抛出一个异常。当然我们也可以使用其他方式来处理这些错误。例如，返回 `-1` 表明没有可用的长度，或者返回一个 `int` 和一个 `error` 值。

毫无疑问，Go 语言的反射包是极其灵活的，允许我们在运行状态做很多事情。但是，引用 Rob Pike 的一句话 [3]：“对于这种强大的工具，我们应当谨慎地使用它们，除非有绝对的必要。”

9.5 练习

本章有3个相互关联的练习，第一个练习要求创建一个自定义的包，第二个练习要求为这个包创建一个测试用例，第三个练习就是利用这个包来写一个程序。这3个练习的难度不断增加，尤其最后一个，很具有挑战性。

（1）创建一个包，比如叫 `my_linkutil`（在文件 `my_linkutil/my_linkutil.go` 里）。同时这个包必须提供两个函数，第一个是 `LinksFromURL(string) ([]string, error)`，给定一个 URL 字符串（如 `http://www.qtrac.eu/index.html`），然后返回这个页面上消重后的所有链接（也就是标签的 `href` 属性值）和 `nil`（或者返回 `nil` 和一个 `error`，如果有错误发生的话）。第二个函数是 `LinksFromReader(io.Reader) ([]string, error)`，也是做相同的事情，只是从 `io.Reader` 里读取数据，比如可能是文件，或者一个 `http.Response.Body`。`LinksFromURL()` 函数可以调用 `LinksFromReader()` 来完成大部分功能。

参考答案在 `linkcheck/linkutil/linkutil.go` 文件里，第一个函数大约 11 行代码，使用了 `net/http` 包的 `http.Get()` 函数，第二个函数大概是 16 行代码，使用了 `regexp.Regexp.FindAllSubmatch()` 函数。

(2) Go标准库提供了HTTP测试的支持（例如`net/http/httptest`包），不过我们这个练习只需测试第一题开发的`my_linkutil.LinksFromReader()`函数。为该目的，请创建一个测试文件，比如 `my_linkutil/my_linkutil_test.go`，包含一个测试用例 `TestLinksFromReader (*testing.T)`。该测试从本地文件系统上读取一个HTML文件和一个包含该HTML文件所有唯一链接的链接文件，然后对比`my_linkutil.LinksFromReader()` 函数分析HTML文件的结果和这个链接文件中的链接。

可以复制 `linkcheck/linkutil/index.html` 文件和 `linkcheck/linkutil/index.links` 文件到`my_linkutil`目录，用来当测试程序的数据文件。

参考答案在 `linkcheck/linkutil/linkutil_test.go` 里，答案里的测试函数大约 40行代码左右，使用`sort.Strings()`函数对找到的结果进行排序，并使用`reflect.DeepEqual()`函数将结果与预期的结果进行对比。如果测试失败，会列出不匹配的链接，方便测试人员测试。

(3) 编写一个程序，比如叫`my_linkcheck`，从命令行读取一个URL（可以有`http://`前缀也可以没有），然后检查每个链接是否是有效的。程序可以使用递归，检查每个链接到的页面，但是不检查非HTTP链接、非HTTP文件及外部网站的链接。应该使用一个独立的goroutine来检查一个页面，这样能实现并发的网络访问，比顺序的一个一个来要快得多了。自然地，可能有多个页面都包含相同的链接，但我们只需要检查一次。这个程序应该使用第一道练习开发的`my_linkutil`包。

参考答案在`linkcheck/linkcheck.go`里，大约150行代码。为了避免检查重复的链接，参考答案里使用了一个映射来维护所有检查过了的URL 列表。这个映射在一个独立的goroutine里维护，并使用3个信道来和它通信：一个用于增加URL，一个用于查询URL是否存在，一个用于返回查询结果。（另外一种方法就是使用第7章的`safemap`。）下面是

一个从命令行输入linkcheck www.qtrac.eu的结果（其中有些行已经被全部或部分的删除）。

```
+ read http://www.qtrac.eu
...
+ read http://www.qtrac.eu/gobook.html
+ read http://www.qtrac.eu/gobook-errata.html
...
+ read http://www.qtrac.eu/comparepdf.html
+ read http://www.qtrac.eu/index.html
...
+ links on http://www.qtrac.eu/index.html
+
checked
http://ptgmedia.pearsoncmg.com/.../python/python2python3.pdf
+ checked http://www.froglogic.com
- can't check non-http link: mailto:someone@somewhere.com
+ checked http://savannah.nongnu.org/projects/lout/
+ read http://www.qtrac.eu/py3book-errata.html
+ links on http://www.qtrac.eu
+ checked http://endsoftpatents.org/innovating-without-patents
+ links on http://www.qtrac.eu/gobook.html
+ checked http://golang.org
+ checked http://www.qtrac.eu/gobook.html#eg
+
checked
http://www.informit.com/store/product.aspx?
isbn=0321680561
+ checked http://safari.informit.com/9780321680563
+ checked http://www.qtrac.eu/gobook.tar.gz
+ checked http://www.qtrac.eu/gobook.zip
```

```
- can't check non-http link: ftp://ftp.cs.usyd.edu.au/jeff/lout/  
+ checked http://safari.informit.com/9780132764100  
+ checked http://www.qtrac.eu/gobook.html#toc  
+ checked http://www.informit.com/store/product.aspx?  
isbn=0321774637  
...
```

[\[1\].文档截屏展示了本书写作时的godoc的HTML渲染结果，现在可能已经发生了改变。](#)

[\[2\].旧版本的Go语言还包含有container/vector包。现在这个包被废弃了，可以使用切片和内置的append\(\)函数。](#)

[\[3\].关于Go语言的反射，Rob Pike写了一篇有趣而实用的博客，见blog.golang.org/2011/09/laws-of-reflection.html。](#)

附录A 后记

Go语言的作者们对一些主流的编程语言进行了深刻的反思，试图识别哪些语言特性是有价值的和有助于提高生产率的，以及哪些特性是多余的甚至是降低生产率的。再基于他们加起来已经有好几十年的编程经验进行总结分析，最终产生了全新的Go编程语言。

与传统的Objective-C和C++相比，Go语言是面向对象的“更好的C”。像Java一样，Go语言有自己的语法，所以它不必像Objective-C和C++那样来兼容C语法。但是和Java不同的是，Go语言是静态编译的，因此也不会受限于虚拟机的速度。

Go语言除了以抽象接口类型和优雅的支持聚合和嵌入的结构类型支持面向对象的全新方式外，也支持函数字面量和闭包等高级特性。同时Go语言内置的映射和切片能够满足绝大多数数据结构的需要。Go语言的Unicode字符串类型使用行业的事实编码标准UTF-8，而且标准库完美支持字节流和字符。

Go语言的并发支持是非常优秀的，它使用轻量级的goroutine和类型安全通道（和锁等不同的是，通道不是底层的数据结构）。与其他的编程语言（如C、C++、Java等）相比，在Go语言里创建并发程序要容易得多。而且Go语言闪电般的编译速度特别适合那些构建大型C++项目或者库的开发人员。

目前Go语言已经被商业或非商业组织广泛使用，Google内部也使用Go语言，，Google App Engine (code.google.com/appengine/docs/go/

overview.html) 上已可以使用Go语言开发Web应用，之前只支持Java和Python。

这门语言目前仍然在快速进化，不过因为有go fix这样的工具，我们可以很容易地将现有的代码升级到最新版本的Go语言。而且，Go语言开发者打算让所有Go语言的1.x版本向后兼容1.0版本，以使Go用户能够拥有一门又稳定又在持续进步的开发语言。

Go语言的标准库非常广泛，但即使它也不满足我们的需求时，我们还可以看看 Go Dashboard (godashboard.appspot.com/project) 能否找到我们需要的，或者我们可以使用其他语言编写的第三方库。要了解Go语言的最新消息可参考golang.org，这个网站有最新的文档、语言规范（很容易看懂）、Go Dashboard、博客、视频和一些其他支持文档。

大部分学习Go语言的程序员都有一些其他编程语言的背景，例如C++、Java、Python等，因此在学习Go语言时通常都已经形成了基于继承模型的面向对象思维。Go语言刻意地不支持继承，所以通常在C++或者Java之间进行代码转换时相对容易，但如果要转换为Go语言，我们最好回到最开始去理解这段代码的目的是完成什么，而不是当前是怎么做的，然后再用Go语言完全重写。也许最重要的不同之处在于支持继承的语言允许将代码和数据混合在一起，而Go语言强制它们分离。分离的好处就是提供了极大的灵活性，更适合于创建并发程序，这对于那些从支持继承的语言过来的程序员，可能要花费一些时间和实践来适应。Go语言的一位核心开发者Russ Cox说：

“很不幸的是，每次有人问我关于继承的问题的时候，我总是回答‘可以啊，使用嵌入就行’。其实嵌入很有用，也是继承的一种方式，但是很多人都没想到这一点。我的看法是：你还在用C++、Python、Java、Eiffel或其他的语言思考方式，停下来，用Go语言的方式思考。”

Go语言是一种学习和使用起来都很令人着迷的编程语言，编写Go语言的代码是一种享受。Go语言开发者会发现加入Go邮件列表对他们很有帮助，因为这个列表拥有很多优秀的发言者，是最适合讨论和咨询问题的地方（[groups.google.com/group/ golang-nuts](https://groups.google.com/group/golang-nuts)）。由于Go语言以开源项目的方式运作，你也可以选择成为 Go语言开发者，帮助维护、改进和扩展 Go语言本身（golang.org/doc/contribute.html）。

附录B 软件专利的危害

专利是资本主义经济中的一种反常现象，因为他们是国家授予的私营垄断。亚当·斯密在他的《国富论》一书中对垄断进行了强烈的谴责。

专利在近现代受到各种商业组织的大范围支持，小到真空吸尘器制造商，大到到制药业巨头。但是谈到软件专利，我们却很难找到到底谁在支持它们，除了那些专利投机者（指那些专门购买或者租赁专利的公司，他们自身从来不创造新东西）和它们的律师。比尔·盖茨曾在1991年说过：“如果在今天的大部分想法被发明的时代人们就已经知道如何进行专利授权，并且申请了相关的专利，那么今天这个行业将处于完全停滞的状态。”当然，他的这个观点今天再听起来有几分微妙。

软件专利影响了每一个生产软件的商业组织，不管生产的软件是用于出售的还是内部使用的。甚至很多非软件行业巨头（像卡夫食品和福特汽车）都不得不花费大笔金钱来对抗软件专利诉讼。但是每个程序员都面临这样的风险。例如，链表也是有专利的，但这个专利并不属于它的发明者Allen Newell、Cliff Shaw和Herbert Simon，尽管这几个人在1955年左右就提出了这个观点，但是50年后却成了别人的专利。skip-list也是同样的情况，由William Pugh在1990发明，却被别人于10年后申请了专利。遗憾的是，有成千上万的软件专利可以用来作为例子，不过这里我们就只再多举一个例子，“一种触发计算机对计算机数据中发现的数据结构进行检测和响应的系统和方法”（A system

and method causes a computer to detect and perform actions on structures identified in computer data)，这个苹果于1999年获得的专利涵盖了所有处理数据结构的软件（www.google.com/patents?id=aFEWAAAAEBAJ&dq=5,946,647）。

我们很容易认为那些范围过广、内容太过泛泛或无法度量的专利应该会比较容易被判断为无效，但实际上，即使是Google这样的巨头也需要花费上百万美金的律师费来保护他们自己。那些初创公司以及中小企业又如何能做到在提供创新软件的同时不被那些像寄生虫一样活着的专利投机者一次又一次地勒索？

在美国以及其他的国家，专利系统大致都是这么运作的。首先，要记住一点，无论当事人是否知道某个专利的存在，该专利都在起作用。而且侵权专利有可能导致天价罚款。现在假设有一个程序员正在完全独立地开发一个闭源软件，设计了一个智能算法来做一些事情。这时有个专利投机者听到小道消息说这个程序员的公司拥有一个可能赚钱的创新。于是，这个专利投机者弄出一个针对该程序员的禁令，宣称说他侵犯了他们的某项泛泛而谈的专利（如上面刚提过的那项苹果专利）。于是，你必须提交你的源码以供独立分析。自然，这个分析将不会仅仅包含引用的专利，而是该专利投机者所拥有的所有专利。这样专利投机者就能完全看到这个程序员所开发的那个智能算法了，他们甚至会考虑自己为这个算法申请一个专利，毕竟他们有足够的钱用于律师费的开销，而钱恰是中小企业所缺的。当然，这些专利投机者不是真的想上法庭，他们的“商业模式”不过是“敲诈勒索”罢了：他们希望这个开发者继续出售他们的产品，并为他们那个宣称被侵犯的某专利支付许可费用。不过，因为绝大部分的专利都无法强制执行，从而无法让专利投机者得到真正的好处，但法庭争斗将导致这些中小企业的大多数破产，因此通常只能以这些中小企业同意支付专

利授权费用而告终。而一旦这些企业继续支付许可费用，专利投机者就可以用这个变得更长的专利授权列表来敲诈下一个受害者。

大公司有能力购买专利并与这些专利投机者对抗，不过他们也没有必要对这些受害的中小企业表现出多少同情，因为这些中小企业可能会成为潜在的竞争对手，所以这些中小企业大部分都不怎么受关注。包括开源公司**Red Hat**在内，一些公司会申请多项专利以作为防御措施，他们可以通过使用专利的相互许可协议来最小化法律费用。但对于像苹果、谷歌和微软这样已经花费数十亿美元构建专利组合的巨头而言，这种做法有多少效果就不得而知了。这些已经花巨资在专利上的巨头不太可能期望这种专利制度被终止（无论这种制度的破坏性有多大），因为制度的终止将导致他们大幅度的账面减记，这对于公司股东和**CEO**们都是不可接受的，因为**CEO**们能得到多少报酬完全取决于公司的股价。

中小企业和独立创新者通常没有足够的资金来抵御专利投机者的勒索。一小部分人会尝试转移到海外，而绝大部分将要么不得不支付大笔费用来抵制这种勒索（或者中途倒闭），要么乖乖为那些没什么价值的专利支付授权费用。软件专利已经对美国的独立创新者和中小企业进行的软件创新事业造成了一定程度的寒冬效应，让这些商业组织的成本变得更高昂，日子过得更艰难，因此也降低了他们为程序员创造更多工作岗位的能力。当然，很多律师从软件专利里得到了不少好处，仅2008年就达到了112亿美元，尽管专业经济学家似乎也无法从软件专利中计算出哪怕一分钱的经济利益。

其他国家的软件开发者也没好到哪里去，有些公司为了避免专利勒索已经不得不将自己的软件从美国市场撤出。这意味着一些创新软件在美国不再可用，从而可能为一些非美国商业组织带去一些竞争优势。不仅如此，**ACTA**（**Anti-Counterfeiting Trade Agreement**，防伪贸易协定）覆盖了所有类型的专利，这个协定正在被全世界包括欧盟在

内逐步接受，但是在我写这本书时还未包括巴西、中国和俄罗斯。此外，欧洲自己的“欧盟专利”（www.unitary-patent.eu）组织很有可能会在整个欧盟范围内实行美国风格的专利制度。

软件是一种受版权严格保护的知识产权。（例如，比尔·盖茨曾经完全靠软件版权成为世界上最富有的人，在软件专利这种恶性创意出现之前就已经做到。）撇开软件版权的成功之处，美国和其他的国家已经开始（或者已经被国际贸易协定强制规定）将软件纳入了它们的专利制度范畴。让我们来想象一下，如果所有这些没什么价值的专利（如范围过广、内容泛泛或是先前技术的专利）忽然消失了会怎么样呢？这必然会大量减少专利勒索并刺激创新。但那个关键疑问依然存在：软件真的需要专利化吗？

包括美国在内的大部分国家不可能会申请与数学公式相关的专利，无论这些公式的新旧。然而，数学公式代表的是想法（也就是专利要“保护”的对象），如我们从“邱奇-图灵论题”中所知，软件的逻辑都可以简化成一个数学公式，所以软件可以归结为数学的一种特定表现形式。这恰恰是Donald Knuth反对软件专利时的论点。（请查阅Knuth教授写的一封简明扼要的信，见www.progfree.org/Patents/knuth-to-pto.txt。）

这个问题也不是不可解决，不过需要立法禁止任何人申请软件专利（或者将软件归类成数学这个不准申请专利的类别，其实我一直认为软件就是数学），而且还需要搞定专利局（专利局会希望专利越多越好，因为他们的收入通常跟专利的数量而非价值成正比）。但这比较困难，因为获取政客的关注需要付出不小的代价，而且那些有能力购买专利的家伙当然也有能力反向游说政客。而且这个主题非常枯燥且行业相关，对那些政客的政治生涯没有太多好处。但真的有很多人在同时游说两个党派以改变现状。你可以从 endsoftpatents.org（在欧

洲是www.nosoftwarepatents.com) 了解更多为什么软件专利具有那么大的灾难性，以及如何打败它们。

附录C 精选书目

Advanced Programming in the UNIX®Environment, Second Edition

W.Richard Stevens and Stephen A.Rago (Addison-Wesley, 2005, ISBN-13: 978-0-201-43307-4)

中文版书名：《Unix环境高级编程》

一本深入详尽地介绍在Unix系统上使用Unix系统调用API和C标准库进行编程的书籍。（书中所有的代码都是用C写的。）

The Art of Multiprocessor Programming

Maurice Herlihy and Nir Shavit (Morgan Kaufmann, 2008, ISBN-13: 978-0-12-370591-4)

中文版书名：《多处理器编程的艺术》

这本书详细地介绍了最底层的多线程编程，对每一个关键技术都提供了优雅和完整的工作代码样例。

Clean Code: A Handbook of Agile Software Craftsmanship

Robert C.Martin (Prentice Hall, 2009, ISBN-13: 978-0-13-235088-4)

中文版书名：《代码整洁之道：敏捷软件开发技能手册》

这本书提出了很多战术性的编程技巧，例如好的命名习惯、函数设计、重构，等等。除此之外，还有很多非常实用和有趣的想法，能够帮助程序员改善他们的编码风格和让代码变得更加易于维护。（这本书的例子都是用Java写的。）

Code Complete: A Practical Handbook of Software Construction, Second Edition

Steve McConnell (Microsoft Press, 2004, ISBN-13: 978-0-7356-1967-8)

中文版书名：《代码大全》

本书旨在如何创建高质量的软件，超越语言特定的领域思想、原理和实践。让程序员深刻思考自己的编程。

Design Patterns: Elements of Reusable Object-Oriented Software

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
(Addison-Wesley, 1995, ISBN-13: 978-0-201-63361-0)

中文版书名：《设计模式：可复用面向对象软件的基础》

现代最具影响力的一本编程书籍，但要消化这本书的内容并不容易。这些设计模式是非常迷人的，而且在我们每天的编程实践中都会用得到。

Domain-Driven Design: Tackling Complexity in the Heart of Software

Eric Evans (Addison-Wesley, 2004, ISBN-13: 978-0-321-12521-7)

中文版书名：《领域驱动设计》

这是一本关于软件设计方面的书，非常有趣，特别是对那些多人参与的大型项目很有用。本书主要是关于创建和改进领域模型（所谓领域模型，是用来表示系统设计目的的），并且还创建了一门贯穿整个系统各个方面的语言，也就是说，本书的读者并不限于软件工程师。

Don't Make Me Think!: A Common Sense Approach to Web Usability, Second Edition

Steve Krug (New Riders, 2006, ISBN-13: 978-0-321-34475-5)

中文版书名：《点石成金：访客至上的网页设计秘笈》

一本关于Web方面的简短、有趣，而且非常具有实践性的书籍，其中给出了许多研究成果和实践经验，对提升网页设计很有帮助。

Linux Programming by Example: The Fundamentals

Arnold Robbins (Prentice Hall, 2004, ISBN-13: 978-0-13-142964-2)

中文版书名：《Linux程序设计》

一本介绍使用Linux系统调用进行Linux程序设计的书籍，实用，而且浅显易懂。（书中所有的代码都是用C语言写的。）

Mastering Regular Expressions, Third Edition

Jeffrey E.F.Friedl (O'Reilly, 2006, ISBN-13: 978-0-596-52812-6)

中文版书名：《精通正则表达式（第3版）》

这本书主要是介绍正则表达式，非常有趣而且有用。